版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用



区块链2.0实战

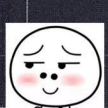
以太坊+Solidity编程从入门到精通

黄振东◎著

不做区块链知识碎片的搬运工，成为知识体系应用的区块链大咖！

系统： 囊括区块链4个层次、12个模块、100+个知识点
专业： 5年区块链研究经验，紧盯全球主流区块链开发
深入： 从宏观到微观，从总体到局部，深度解码硬核知识
前沿： 超过40组官方推荐源代码，借鉴主流通用编程案例

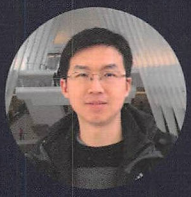
中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

作者简介



黄振东

经济管理学硕士，上市公司海外投资总监，长期关注全球金融、商业与投资领域的发展趋势，专注于金融、投资并购和互联网科技领域的前沿探索，应用深厚的理论框架和实践经验进行解读和分析，致力于在理论和实际操作层面掌握创新金融科技。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

区块链2.0实战

以太坊+Solidity编程从入门到精通

黄振东◎著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



内 容 简 介

网络上的各种区块链知识虽然很多，但难以构成体系，本书根据读者的需求，以完整的体系介绍了当前主流的区块链技术。本书从总体出发，介绍了区块链的发展历程、典型应用、在社会生活各方面的应用前景，并进一步深入基础技术层面，详细介绍了区块链技术的各个组成部分，最后介绍了区块链 2.0——以太坊的发展情况和开发编程，引导读者由远及近、从感性到理性、从原理到实践，建立起全面的区块链知识体系，逐步深入地掌握区块链技术，并具备应用以太坊编程语言开发分布式应用程序的初步能力。

本书以全球化的眼光专注于区块链技术的体系搭建，定位高端、专业性强、内容全面、便于实操，既适合金融行业的投资人员研究和掌握区块链的技术与商业价值，也适合想在区块链领域有所发展的创业者学习，还可以作为编程开发人员的参考资料，同时适合想了解区块链技术的大专院校学生作为学习资料。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

区块链 2.0 实战：以太坊+Solidity 编程从入门到精通 / 黄振东著. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34877-8

Ⅰ. ①区… Ⅱ. ①黄… Ⅲ. ①电子商务－支付方式 Ⅳ. ①F713.361.3

中国版本图书馆 CIP 数据核字（2018）第 184612 号

策划编辑：徐　岩
责任编辑：张　毅　　特约编辑：田学清
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　　印张：22.5　　字数：468 千字
版　　次：2018 年 10 月第 1 版
印　　次：2018 年 10 月第 1 次印刷
定　　价：78.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei. com.cn。

本书咨询联系方式：（010）57565805。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前言

从2017年下半年以来，区块链技术在全世界掀起了又一轮高潮，无论是金融行业的从业人员、投资专家，还是普通的股民，甚至从未接触过金融科技的人士，在网络、移动终端上都能感受到区块链的热浪。2018年春节期间，不眠不休的“三点钟投资群”再次点燃了许多人的热情。更不用说各种虚拟货币的币值剧烈波动，吸引了大量的资金投入进来。在全社会各个阶层的人们越来越多地投入区块链领域时，各个国家和地区也在加强对区块链的监管，比如中国央行等7部委在2017年9月4日专门出台政策规范区块链领域的融资乱象，2018年1月中国央行再次强化对区块链领域虚拟货币的监管政策。进入2018年中期，区块链带给全社会的热潮逐渐退去，虚拟货币的币值也逐步向价值回归，在这一轮炒作过去之后，区块链留给我们的更多的是思考和行动。

毋庸讳言，区块链技术是一种具备强大生命力的、可以改变现有商业规则的新技术。在一波又一波的热潮洗礼下，许多人了解到区块链的一些特点，如去中心化、分布式账本、不对称加密及无法篡改等，也能说出一些基本道理。但是如果再深入思考：到底什么是区块链技术？它的技术架构和技术体系是怎样的？面对这样体系化的问题，人们如果仅凭自己从社交媒体上获得的关于区块链的碎片式知识，那么显然是难以回答的，进而也难以在认知上建立起体系化的区块链知识结构。

另外，从2009年1月比特币的首次出现到现在，区块链技术进入人类社会已经过去了近10年，这期间虽然有所曲折，但区块链技术在全球总的发展态势还是在向人们生活的各个层面不断深入。以软件版本进行类比，区块链已经走过了1.0阶段，区块链2.0也已经成为很多从业人士的日常工作内容，区块链

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.0 正在向普通人走来。在这样一个时点上，如果想要了解区块链，除从原理和基础知识方面来掌握以外，更重要的还是要进入操作层面，从区块链 2.0 的以太坊应用开发入手，在具体实践中获得对于区块链的全新认知。正如古语有云：纸上得来终觉浅，绝知此事要躬行。

从这一角度出发，本书首先选择了“体系化的知识结构”和“可操作性的开发指南”两个层面，为读者构建了区块链的知识体系，包括区块链的总体架构、发展历程、系统构造、基础技术等方面的内容，让读者对于区块链技术建立一个从宏观到微观的总体结构性认知；然后引领读者转入对区块链 2.0——以太坊的深入了解过程，在区块链公有知识的基础上认识以太坊的独特之处，并由此进入对以太坊主流语言 Solidity 的介绍，在了解了 Solidity 语言的基础知识之后，以鲜活的编程实践引导读者灵活运用 Solidity 语言开发出自己需要的区块链分布式应用（DApp）。

区块链是一项前沿的金融科技，为此，本书坚持全球化的视野，很多内容直接采用欧美区块链行业的信息，在编程开发的案例中也选用更新日期最近的开发案例，突出地展示了本书注重时效性、紧盯科技前沿的鲜明特点。在写作方式上，本书采用了大量的图表，以图文并茂的方式增强可读性；全书行文朴实自然，力求表述准确，让读者可以清晰地掌握区块链这种具备较强数理特性的技术。

总之，关于区块链，我们已经讨论了很多，现在是时候进入操作阶段了。

因受作者水平和成书时间所限，本书难免存在疏漏和不当之处，敬请指正。

本书特色

1. 内容全面、结构清晰，有助于读者建立对区块链的系统性认知体系

本书系统而全面地介绍了区块链技术的知识体系，从宏观到微观、从总体到局部、从感性认知到理性知识，有利于读者建立全面的区块链知识体系，不再局限于碎片式阅读产生的点状知识图，符合人们从浅层学习提升到深入思考的要求。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2．行文朴实自然、表述准确，有利于促进读者对区块链技术的理解

本书介绍的区块链知识体系、原理，以及采用以太坊编程开发的内容都采用朴实自然的语言，在兼顾专业性、准确性的前提下，尽量做到行文通俗易懂，以便于读者阅读和理解，提升对于区块链的掌握水平。

3．充分使用图表方式，将抽象事物形象化

由于区块链技术是一种数理性较强的技术，因此本书在介绍一些过于专业的知识点时，使用大量的图表来补充说明，通过这种方式使原本专业性很强的知识或信息转变为清晰、易理解的内容，增强了本书的可读性。

本书内容及体系结构

第 1 章　区块链基础

本章从宏观方面对区块链技术进行“空间扫描”，为读者呈现了区块链技术的总体图像，解释了区块链的定义，回顾了区块链技术在全球的发展历程和现状，着重介绍了区块链在中国的发展现状及发展前景。

第 2 章　区块链的商业价值

本章以区块链在国内商业环境下的 5 个典型行业——银行业、电子商务、法律行业、影音娱乐和媒体行业、医疗行业的商业价值，分析了区块链对于解决当前这些行业“痛点”、提升效率的价值，点明了区块链的商业价值。

第 3 章　区块链技术原理

本章首先介绍了区块链技术最成熟应用——比特币的技术原理、组成部分和运行机制，由此提炼并引申介绍了区块链的技术原理、系统框架、分布式网络等基础原理。

第 4 章　区块链共识机制

本章介绍了区块链技术最重要的运行机制——共识机制，并逐一介绍了当前主流的 3 种共识机制的详细情况。此外，还介绍了区块链在不同规模等级下

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



的 3 种区块链类型，以及各自的应用案例。

第 5 章　区块链数据结构

本章介绍了区块链的数据结构、区块链技术所依赖的核心算法——哈希算法，并深入介绍了区块链不对称加密所用的椭圆曲线加密算法，在此基础上介绍了区块链的运行机制，以及怎样对区块中的数据进行解读。

第 6 章　以太坊

前 5 章为读者搭建了区块链技术的总体了解和完整体系，从第 6 章开始将深入区块链 2.0 的世界，对以太坊区块链进行深入的讲解。本章介绍了以太坊的创立和发展过程，并着重介绍了以太坊的各项技术原理。

第 7 章　以太坊应用开发基础

为了促进读者对于以太坊的实践应用，本章聚焦于介绍以太坊应用开发基础，主要介绍了 5 种常见的以太坊应用开发环境，对每种应用开发环境则介绍了其搭建的具体操作过程。另外，本章还引入了对主流以太坊编程语言——Solidity 语言的初步介绍。

第 8 章　Solidity 开发基础

本章介绍了 Solidity 语言的各种基础知识，为读者应用 Solidity 语言进行分布式应用开发打下了坚实的基础。主要介绍了 Solidity 语言的 17 种数据类型和 10 种控制结构，还着重介绍了以太坊合约和 Solidity 汇编过程，其中以太坊合约是区块链分布式应用（DApp）的核心要素。

第 9 章　采用 Solidity 语言开发以太坊游戏

本章介绍了一个采用 Solidity 语言开发以太坊游戏的案例，以此来引导读者灵活应用 Solidity 语言的各种基础元素开发自己的应用程序。其中介绍了以太坊游戏的特点、开发准备，还介绍了以太坊游戏 Influence 的代码框架，并对其主要源代码进行了分析和解读。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



本书读者对象

- 从事区块链项目投资的各类投资机构从业人员。
- 希望进行区块链应用开发的软件开发人员。
- 希望利用区块链开发创新业务的金融机构从业人员。
- 大专院校金融或互联网科技等相关专业的学生。
- 从事区块链研究的专家学者。
- 其他对区块链感兴趣的各类人员。



仅供非商业用途或交流学习使用

目录

第 1 章　区块链基础 1

1.1　什么是区块链 1

1.1.1　区块链的定义 2

1.1.2　区块链的运行流程和特点 5

1.1.3　区块链的类型 6

1.2　区块链的发展经过和现状 7

1.2.1　区块链产生的背景 7

1.2.2　比特币的诞生 9

1.2.3　比特币的底层技术是区块链 11

1.2.4　区块链全球发展的不均衡特点 12

1.2.5　区块链在全球的发展现状 13

1.3　区块链在中国的发展现状 18

1.3.1　重视底层突破，区块链技术创新加速 18

1.3.2　资本快速进入，区块链融资增长迅猛 18

1.3.3　全产业链布局，区块链应用领域逐步拓展 19

1.3.4　抱团发展，各类区块链行业组织纷纷成立 19

1.4　中国区块链行业的发展前景 20

1.4.1　“90 后”创业者人群大量入场 20

1.4.2　大型企业积极参与，区块链技术基础更加深厚 20

1.4.3　全国各地高度支持区块链发展 21

第 2 章　区块链的商业价值 23

2.1　区块链在银行业的应用 23

仅供非商业用途或交流学习使用

2.1.1 区块链对银行业的改变 24
2.1.2 国内外银行业的区块链应用 24
2.2 区块链与电子商务 27
2.3 区块链在法律行业的应用 29
2.3.1 区块链证据 30
2.3.2 智能合约 30
2.3.3 区块链权证 31
2.4 区块链在影音娱乐和媒体行业的应用 32
2.5 区块链在医疗行业的应用 34
2.6 结语 36
第 3 章 区块链技术原理 37
3.1 比特币带来了区块链 37
3.2 比特币白皮书 39
3.2.1 简介 39
3.2.2 交易 40
3.2.3 时间戳服务器 41
3.2.4 工作量证明 41
3.2.5 网络 43
3.2.6 激励 44
3.2.7 回收硬盘空间 44
3.2.8 简化的支付确认 45
3.2.9 价值的组合与分割 46
3.2.10 隐私 47
3.2.11 计算 47
3.2.12 结论 50
3.3 比特币系统的参与者 51
3.4 比特币区块 54
3.4.1 比特币的交易过程 54
3.4.2 比特币挖矿 55
3.5 长链与短链 57

3.6 比特币的安全性.....59
3.7 比特币挖矿设备的发展.....60
3.7.1 比特币挖矿设备的发展阶段.....61
3.7.2 矿机与矿场.....62
3.7.3 矿池.....64
3.7.4 云挖矿的应用.....66
3.8 比特币交易中的非对称加密.....67
3.8.1 非对称加密原理.....67
3.8.2 生成钱包地址.....70
3.8.3 交易加密过程.....71
3.9 从比特币到区块链.....73
3.10 区块链的系统框架.....75
3.11 分布式网络.....77
3.11.1 分布式网络的概念.....77
3.11.2 分布式网络的特点.....78
3.11.3 分布式网络的两种架构.....79
3.12 广播与验证机制.....81
3.12.1 广播.....82
3.12.2 验证.....82
第 4 章 区块链共识机制.....85
4.1 共识机制的意义.....85
4.2 工作量证明机制.....86
4.2.1 哈希函数.....87
4.2.2 工作量证明机制的基本原理.....88
4.2.3 比特币的工作量证明过程.....89
4.2.4 工作量证明机制的优缺点.....92
4.3 权益证明机制.....93
4.3.1 PoS 机制与 PoW 机制的区别.....93
4.3.2 PoS 区块创建.....96
4.3.3 PoS 机制的发展过程.....97

4.4 授权股权证明机制 98
4.4.1 DPoS 概述 99
4.4.2 选举见证人 99
4.4.3 选举授权代表 100
4.5 区块链的三大类型 101
4.5.1 公有链 101
4.5.2 私有链 104
4.5.3 联盟链 105
第 5 章 区块链数据结构 109
5.1 区块链的数据组成 109
5.1.1 区块的数据结构 110
5.1.2 区块链数据结构的技术基础 111
5.1.3 哈希算法 112
5.1.4 默克树 113
5.1.5 时间戳 116
5.1.6 难度目标 117
5.1.7 随机数 118
5.2 区块链技术的算法 119
5.2.1 哈希算法概况 120
5.2.2 SHA256 算法 122
5.3 椭圆曲线加密算法 129
5.3.1 椭圆曲线加密算法的特点 129
5.3.2 椭圆曲线加密算法的数学原理 130
5.3.3 椭圆曲线加密算法的加密原理 136
5.4 区块链的运行机制 138
5.5 区块链的交易机制 139
5.5.1 交易流程 139
5.5.2 比特币钱包 141
5.5.3 交易身份验证 142
5.6 区块链的造链机制 144

5.6.1 验证接收信息 144
5.6.2 创建区块 147
5.7 区块数据解读 152
第 6 章 以太坊 161
6.1 以太坊概述 161
6.2 以太坊的创立和发展 162
6.3 以太坊技术原理 165
6.3.1 以太坊与比特币的联系 165
6.3.2 以太坊账户 167
6.3.3 交易和消息 168
6.3.4 燃料 169
6.3.5 合约 171
6.3.6 智能合约示例 177
6.3.7 以太坊挖矿 179
6.3.8 以太坊区块 183
第 7 章 以太坊应用开发基础 195
7.1 以太坊开发环境的建立 195
7.2 Geth 开发环境 198
7.2.1 初次启动 198
7.2.2 命令行安装模式 199
7.2.3 Geth 的使用 200
7.2.4 Geth 在私有链上的应用 202
7.3 轻节点模式 207
7.3.1 Ganache 图形界面 207
7.3.2 安装 Truffle 209
7.3.3 运行 Ganache 211
7.4 网页模式 213
7.5 Mist 浏览器 217
7.5.1 Mist 安装 218
7.5.2 Mist 应用 221

7.6 用 MetaMask 建立开发账户 225
7.7 以太坊开发的编程语言 Solidity 229
7.7.1 Solidity 语言简介 230
7.7.2 Solidity 语言的常用语句 230
7.7.3 Solidity 程序初步解读 236
第 8 章 Solidity 开发基础 239
8.1 Solidity 语言的数据类型 240
8.1.1 数值类型 240
8.1.2 地址类型 242
8.1.3 字节数组 244
8.1.4 地址常量 245
8.1.5 有理数和整数常量 245
8.1.6 字符串常量 246
8.1.7 十六进制常量 247
8.1.8 枚举类型 247
8.1.9 函数类型 248
8.1.10 数据位置 251
8.1.11 数组 252
8.1.12 结构 255
8.1.13 映射 257
8.1.14 包含左值的运算符 258
8.1.15 删除 258
8.1.16 基本类型的转换 259
8.1.17 类型推导 260
8.2 Solidity 语言的表达式和控制结构 261
8.2.1 函数的输入参数和输出参数 261
8.2.2 控制结构 262
8.2.3 返回多元值 262
8.2.4 函数调用 262
8.2.5 函数参数的显名调用 263

8.2.6 省略函数参数的名称 264
8.2.7 在合约中创建新合约 264
8.2.8 解构赋值和返回多元值 265
8.2.9 范围和声明 266
8.2.10 错误处理：断言、请求、还原与异常 267
8.3 以太坊合约 270
8.3.1 创建合约 270
8.3.2 可见性 272
8.3.3 取值函数 274
8.3.4 函数修饰符 274
8.3.5 常数状态变量 276
8.3.6 视图函数 277
8.3.7 纯函数 278
8.3.8 后备函数 278
8.3.9 重载函数 279
8.3.10 事件 280
8.3.11 继承 282
8.3.12 构造器 285
8.3.13 抽象合约 286
8.3.14 接口 287
8.3.15 库 287
8.3.16 用于“using for” 291
8.4 Solidity 汇编 292
8.4.1 内联汇编 292
8.4.2 句法 294
8.4.3 操作码 294
8.4.4 访问外部变量和函数 297
8.4.5 本地汇编变量的声明 298
8.4.6 赋值 299
8.4.7 if 语句 299

8.4.8 switch 语句 300
8.4.9 循环 300
8.4.10 函数 301
8.4.11 独立汇编 301
第 9 章 采用 Solidity 语言开发以太坊游戏 304
9.1 以太坊游戏的特点 304
9.2 以太坊游戏开发准备 306
9.3 以太坊游戏 Influence 代码框架 308
9.4 以太坊游戏 Influence 源代码解读 311
9.4.1 游戏界面 312
9.4.2 库文件 lib 314
9.4.3 游戏主功能：小行星拍卖 323
9.4.4 游戏主功能：小行星代币 331
9.4.5 小结 341
后记 深入区块链，用技术改变未来 342

chapter ONE

第1章

区块链基础

区块链已经成为全球范围内的热点。许多人可能被周围环境所影响，看到了大量的资金、人才、信息和资源在向区块链领域集中，或者从社交媒体上感受到了区块链领域追求发展的急切氛围，但是对于什么是区块链这样的基础性认知反而并不清晰。在此，我们先对区块链领域进行一次“空间扫描”，建立起对区块链领域的总体认识。

1.1 什么是区块链

2018 年，距离 2008 年的全球金融危机已经过去了 10 年。10 年以来，随着科技的发展，以互联网、人工智能（Artificial Intelligence，AI）为代表的金融科技（FinTech）蓬勃发展，人类似乎找到了避免全球金融危机的新方法。但是很多专家预计，只要以银行为中心节点的金融体系架构没有发生根本性的转变，危机仍将在未来某个时点再度前来，尽管我们可以有更多的应对之道。

探寻全球金融危机产生的根源，人们逐渐认识到：中心化的全球金融环境是危机产生的根源，只有从根源上解决危机产生的机制，才能够真正引领人类的发展进入新的阶段。在这方面，2009 年伴随着比特币而诞生的区块链技术成为人类新金融、新发展的基础性思维，它彻底颠覆了金融系统数百年来固有的

中心化架构，令改变自根源开始生长，必将全面改变金融系统的生态，进而向人类生活的各个方面演进，全方位地改变人们的生活方式。不同于以往的其他技术创新，区块链代表着一种内源性力量的觉醒。

TIPS:

中国古代先贤老子从“天道（自然规律）”的角度提出了对人类生活具有普遍意义的重大变革的特点：大音希声，大象无形。意思是：越是大的成就往往越穿透悠远，越是大的气度往往越包容万物。因此，从哲学的角度来看，就像“天网恢恢，疏而不漏”一样，区块链必将覆盖人类生活的各个方面，为我们提供新的理念、生活方式及价值。

1.1.1 区块链的定义

那么，究竟什么是区块链？我们在这里从专业到通俗给出解释和说明，从不同的角度阐明区块链的特点。

由于区块链最早产生于美国，所以我们首先看看维基百科的解释。英文维基百科对区块链（Blockchain）的定义是：

A blockchain, originally block chain，is a continuously growing list of records, called blocks, which are linked and secured using cryptography. Each block typically contains a cryptographic hash of the previous block, a timestamp and transaction data. By design, a blockchain is inherently resistant to modification of the data. It is “an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way”. For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks, which requires collusion of the network majority.

这段文字翻译过来就是：区块链是一种不断增长的记录列表，这种记录称为“区块（Block）”，它们链接到一起，并使用密码加密以确保安全。每个区块由数个典型内容组成：前一区块的加密哈希值、时间戳和交易数据。通过设计，

区块可以从本质上阻止对交易数据的篡改。区块链是“可以在交易双方之间以有效、可验证、永久性方式记录交易数据的一种开放、分布式总账”。由于用作分布式总账，所以区块链的典型特点是由点对点网络集体遵循一种协议来认可新区块，一旦新区块被记录下来，只有当网络的大多数节点同意时，区块内的数据才可以由所有链接的区块共同进行修改。

我们认为，从本质上来讲，区块链是一种分布式、去中心化的网络数据库系统，在这个网络中可以发生无数各类交易，所有的交易都由网络的全部节点参与确认和维护，通过共识机制来保证交易与信息的安全性和有效性；与此同时，网络中的全部交易数据以加密形式存储到网络的所有节点上，并通过合理的机制设计来保证系统在不需要中心机构的前提下可追溯与稳定运行。

如果要简单地理解区块链，则可以把它看作一本公开的“总账”，网络的每个节点都保存着这本“总账”的副本，总账的每一“页”就是一个区块，存储着一段时间内的系统内所有交易的加密信息，而每一页都包含着前一页的“页码（前一个区块的地址）”，形成一条时间链和一条物理空间链，从而在理论上使每笔交易都可以追溯至最初的本源。

TIPS:

为什么说是“理论上”？因为从原理上是可以做到的，但真要这样做，则需要大量的时间和资源通过一个个区块去追溯，不具备经济性；一般的情况只追溯、验证区块头的哈希值，就可以等价于该区块的所有交易是可验证的。

由于每个节点都有总账的副本，因此任意一个节点如果想要伪造交易，那么从理论上来说需要修改全网每个节点的总账，一方面时间有限，另一方面计算资源也有限。也就是说，通过伪造区块、伪造历史交易来达成私利在区块链中是很难做到的，强行去做就会消耗巨量的算力来对整条区块链进行篡改。因此，我们可以理性地认为单一节点无法对历史交易进行篡改，从而确保了数据的可靠性和安全性。

而这本总账每次新增加一页（增加一个新的区块），需要通过一种算法取得全网 51%以上节点（网络节点的大多数）的认可才能构成区块链，而不是像传

统的网络或交易那样，通过一个权威机构（网络中心）的认可来增加信用，即由网络共识替代了网络中心，用公开和透明的机制来建立交易信任，根除了网络中心出现失效或舞弊的风险。

为了更加通俗地理解区块链的含义，我们可以从谈恋爱和婚姻的角度来类比什么是区块链。在传统的恋爱、结婚模式下，采用的是“中心化”模式，比如某男和某女恋爱结婚，封建社会以媒妁之言为准，现代社会则需要到民政局婚姻登记中心办理婚姻登记，由婚姻登记中心这一“中心”来对婚姻进行确认。

如果由于某种原因，比如婚姻登记中心的电子档案管理系统被计算机病毒感染导致所有的信息丢失（或者因其他各种情况导致婚姻登记中心的所有资料全部失去），那么，此时即使他们手中持有结婚证，但因为没有了婚姻登记中心的加持和认证，就失去了权威性，可以被改动。对于信用度缺乏、没有良知的“渣男”来说，他完全可以否认本次“结婚”交易的有效性，再去找其他女人结婚（重新交易）；而女方则只能有苦难言，吞下被“渣男”骗婚的苦果……

相反，如果采取的是“去中心化”的模式，即某个男子想要与某个女孩结婚，都需要到一个有成千上万人的广场上进行宣誓，由广场上的成千上万人对他的婚姻进行确认，那么骗婚这种情况就不会发生。

比如，某男为了追求到某女和她结婚，来到广场，面对广场上所有的人发誓：他全心爱着某女，此生只与某女结婚，再也不会与别的女性发生婚姻关系！在该男发出了“婚姻（交易）”信息后，广场上的成千上万人都相信了该男的话，并各自拿出小本子记下了该男的誓言。此后如果该男想要否认这次婚姻，或者再去与别的女性结婚，那么当他再次来到广场发誓时，广场上所有的人都会查询自己的小本子，发现该男已经与某个女孩结婚了，他无法否认上次的婚姻，更不能再次与其他女孩结婚！男子的骗婚行为在“去中心化”的人民群众面前被无情击垮了，彻底保护了真心“交易”的女方的利益。这就是“去中心化”的区块链的力量。

TIPS:

虽然上述故事杜撰的成分居多，但在实际生活中，中国人举办结婚仪式，一般要广泛邀请男女双方的亲朋好友参与婚宴，共同确认双方正式结

婚，这实际上也有一种小范围的“去中心化”共识确认的思想蕴含在其中，可见区块链技术的确隐含了通过共识增加信任的人类社会基本需求。

1.1.2　区块链的运行流程和特点

区块链是一个分布式账本和去中心化数据库，以此来维护连续增长的交易记录（区块）。每个区块中都包含着时序标记（称为时间戳）和指向前一个区块的链接。这也是区块链运行机制的关键点之一，通过每个区块指向前一个区块（父区块）的链接，确保了系统内数据的稳定和不易被篡改的特点。

由于区块链是一个分布式账本和去中心化数据库，所以它没有中心化的权威机构来保证网络的安全与稳定，而由全部网络节点共同维护，也就是说账本内容在每个节点上都有一份副本，每个节点都可以访问账本内容，并参与账本的共同维护过程。对账本内容的更新和确认则是通过所有参与方的共识机制来完成的，即当所有人认可后，交易的内容被写入有效区块，并打上时间戳，添加到按时间顺序线性排列的区块链的末尾。

TIPS:

新的区块与此前的最后一个区块相链接，因此通过访问区块链上各个区块的内容，实际上就可以掌握所有历史交易的信息。

由于每个节点都掌握着账本内容的一份副本，因此任意时刻、任意节点都可以对某笔交易、某个人是否拥有某项财产做出判断和证实；同时，想要篡改交易内容，就变得无比困难，因为要同步改动至少 51%的网络节点所掌握的全部历史数据的区块，这需要巨量的算力才能完成。因此，区块链的安全性是通过群体的参与来实现的。

总的来说，区块链具有如下特点。

（1）去中心化：区块链数据的验证、记账、存储、维护和传输等过程均基于分布式网络结构，采用纯数学方法，而不是依托中心机构来建立分布式节点间的信任关系，从而形成去中心化的、可信任的分布式系统。

（2）时序数据：区块链采用带有时间戳的链式区块结构存储数据，从而为数据增加了时间维度，具有极强的可验证性和可追溯性。

（3）集体维护：区块链系统采用特定的经济激励机制来保证分布式系统中的所有节点均可参与数据区块的验证过程，并通过共识算法来选择特定的节点将新区块添加到区块链；通过集体维护的方式来保证交易数据存储的稳定性和可靠性。

（4）可编程：区块链技术可提供灵活的脚本代码系统，支持用户创建高级的智能合约、数字货币或其他形式的去中心化应用。

（5）安全可信：区块链技术采用非对称密码学原理对数据进行加密，即只能单向加密而不能反向解密，确保了数据和信息的基础安全；同时借助分布式系统，由各节点的工作量证明等共识算法形成的强大算力机制来抵御外部攻击，保证区块链数据不可篡改和不可伪造，因而具有较高的安全性。

1.1.3 区块链的类型

根据区块链的网络范围、开放程度等不同，区块链可以分为 3 种类型：公有区块链、私有区块链及联盟区块链，其中私有区块链和联盟区块链可以被认为是广义的私链。

1. 公有区块链（Public Blockchain）

公有区块链是指世界上任何个体或者团体都可以发送交易，且交易能够获得该区块链的有效确认，任何人都可以参与其共识过程的区块链。公有区块链是最早出现的区块链，也是目前应用最广泛的区块链。目前网络上大量出现的各种虚拟数字货币均基于公有区块链，其中比特币区块链是目前世界上最成功、接受程度最高、最公开的公有区块链。

2. 私有区块链（Private Blockchain）

私有区块链是指在一个相对小的范围内建立的、仅供小范围内应用的区块链。它仅仅使用区块链的总账技术进行记账，可以是一家公司，也可以是个人，独享该区块链的写入权限。在这种情况下，私有区块链与其他的分布式存储方式没有太大的区别。

3. 联盟区块链（Consortium Blockchain）

联盟区块链又被称为“联合（行业）区块链”，在某个群体内部预先选定多

个节点作为记账人，每个区块的生成由所有的预选节点共同决定（只有预选节点才能参与共识的形成过程），其他接入节点可以参与交易，但不过问记账过程（从本质上来看还是托管记账，只是变成了分布式记账，其中，预选节点的数量多少和预选节点如何决定每个区块由谁来记账这两个方面成为该类区块链的主要风险点，即：如果预选节点数量过少，或者预选节点决定某个区块记账者的机制比较单一，则将会造成这类区块链在事实上由少数节点取得控制权，导致少数节点可以篡改交易，最终无法建立信任，联盟区块链也面临崩溃），其他任何人可以通过该区块链开放的 API 进行限定范围的查询。

1.2　区块链的发展经过和现状

区块链并不是一开始就作为一种单一的技术被开发出来的，它是伴随着数字货币——比特币的发明而出现的。准确地说，比特币是人类历史上第一种使用区块链技术的数字货币应用。随着比特币逐步进入人类的生活，区块链也从比特币应用中独立出来，成为人们广泛研究的、具有广阔前景的一种颠覆性技术。要深入了解和掌握区块链技术的发展，数字货币尤其是比特币的产生是一个不可或缺的环节。

1.2.1　区块链产生的背景

进入 21 世纪以来，随着信息技术的发展，人类的生活逐渐网络化、数字化，开发人员把现实生活中的各种场景用信息技术加以优化，提高效率、创新模式，形成了大量的新产品、新技术。

TIPS:

在这种背景下，有很多研究人员开始思考和探索：作为交易的媒介——货币，在人类历史上已经存在了几千年，它在信息技术的融合下会发生怎样的变化？从 20 世纪八九十年代开始，大量的程序员开始研究和探索数字货币。

1993 年，美国人埃里克·休斯和其他一些密码学爱好者创建了一个名为

“密码朋克邮件名单”的加密电子邮件系统，简称“密码朋克”，其实相当于密码学和数字货币研究者的“微信群”。在顶峰时期，密码朋克约有 1400 位用户，参与人士包括当时 IT 领域的许多精英，比如“维基解密”的创始人阿桑奇、提出智能合约概念的尼克·萨博、Facebook 的创始人之一肖恩·帕克、菲利普·希默曼（PGP 技术的开发者，密码朋克的创始者之一）、约翰·吉尔摩（太阳微系统公司的明星员工，也是密码朋克的创始者之一）、斯蒂文·贝洛文（美国贝尔实验室研究员，哥伦比亚大学计算机科学教授）、布拉姆·科恩（BT 下载的作者）、蒂姆希·C·梅（英特尔公司前首席科学家）……这个小型的私人团体是密码天才的松散联盟，成员之间经常就数字货币展开讨论，并将一些想法付诸实践，他们讨论的话题既包括数学、加密技术、计算机技术、政治和哲学，也包括私人问题。据统计，在比特币诞生之前，密码朋克的成员讨论、发明或失败的数字货币和支付系统多达数十个。

亚当·贝克（Adam Back）是英国的一位密码学家，1997 年，他发明了哈希现金（Hash Cash），其中用到了工作量证明（Proof of Work，PoW）机制。这个机制的原型是用于解决互联网垃圾信息问题的，后来被引入区块链原理之中，成为比特币的核心要素之一。

哈伯和斯托尼塔（Haber and Stornetta）在 1997 年提出了一个用时间戳的方法保证数字文件安全的协议，这个协议成为比特币区块链协议的原型。

戴伟（Wei Dai）是一位兴趣广泛的密码学家，他在 1998 年发明了 B-money。B-money 强调点对点的交易和不可更改的交易记录。不过，在 B-money 中，每台计算机单独书写交易记录，这很容易造成系统账本的不一致。戴伟为此设计了复杂的奖惩机制以防止作弊，但是并没有从根本上解决问题。中本聪在发明比特币的时候，借鉴了戴伟的很多设计，并和戴伟有很多邮件交流。

哈尔·芬尼（Hal Finney）是 PGP 公司的一位高级开发人员，也是“密码朋克”运动早期和重要的成员。2004 年，芬尼推出了自己研发的电子货币，在其中采用了可重复使用的工作量证明机制（RPoW）。哈尔·芬尼是第一笔比特币转账的接受者，在比特币发展的早期与中本聪有大量的互动和交流。

正是有了这些密码朋克成员在数字货币领域的探索与实践，为比特币的诞生提供了基础和大量的借鉴。

1.2.2　比特币的诞生

TIPS:

2008 年 9 月，以雷曼兄弟的倒闭为开端，金融危机在美国爆发并向全世界蔓延。为了应对危机，美国政府采取多轮量化宽松等措施，救助由于自身过失而陷入危机的大型金融机构。这些措施带来了广泛的质疑，并一度引发了美国民众“占领华尔街”运动。

2008 年 10 月 31 日纽约时间下午 2 点 10 分，在密码朋克的一个邮件列表中，几百位成员均收到了名为中本聪（Satoshi Nakamoto）的人的电子邮件，“我一直在研究一个新的电子现金系统，这完全是点对点的，无须任何可信的第三方”，然后他给出一个链接，向全体成员公布了一本长达 9 页的白皮书，其中描述了一个新的货币体系。同年 11 月 16 日，中本聪在网络上发出了比特币代码的先行版本。

2009 年 1 月 3 日，中本聪在位于芬兰赫尔辛基的一台小型服务器上挖出了比特币的第一个区块——创世区块（Genesis Block），并获得了“首矿”奖励——50 个比特币。在创世区块中，中本聪写下这样一句话：

“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.”（中文译为：“2009 年 1 月 3 日，财政大臣站在第二次救助银行的边缘。”）这句话是当天《泰晤士报》头版的标题。中本聪将它写进创世区块，不但清晰地展示着比特币的诞生时间，还表达着对旧体系的嘲讽。

TIPS:

尽管比特币的白皮书在 2008 年已经在密码朋克中发表，但是由于创世区块、首次挖矿等实际应用才体现了比特币从理念、设计进入真实的网络世界，所以从这个角度来说，2009 年 1 月 3 日才是比特币的真正“生日”。

2010 年 5 月 18 日，比特币的早期开发人员 Laszlo Hanyecz 在比特币论坛 BitcoinTalk 上发帖声称：“我愿意付 10 000 个比特币来购买两个比萨，大概两个大的就够了，这样我可以吃一个，然后留一个明天吃。你可以自己做比萨送过来，也可以在外面订外卖然后送到我的住址。”他还对比萨的种类提出了要求：“我喜欢放洋葱、胡椒、香肠、蘑菇等的比萨，不需要奇怪的鱼肉比萨。如果你

对此感兴趣请回复，我们可以做一笔交易。”

在帖子发出之后，有几个感兴趣的人陆续做出了回复，除有人问地址外，还有人表示自己在欧洲，不知道如何帮 Laszlo Hanyecz 订美国的外卖。Laszlo Hanyecz 附上了自己在佛罗里达州的地址，有人回复说 10 000 个比特币太少了，在另一个网站上可以换到 41 美元。几天后，也就是 5 月 22 日，Laszlo Hanyecz 发出了帖子，表示已经和一个叫 Jercos 的人完成了交易，还附上了比萨的图片，如图 1.1 所示。

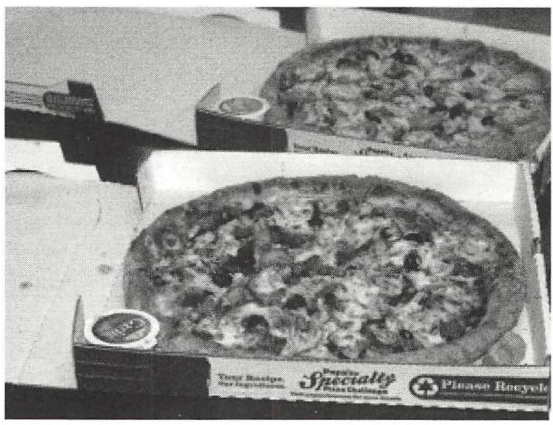


图 1.1　Laszlo Hanyecz 用 10 000 个比特币购买的两个比萨

至此，这个不超过 20 人回复的帖子沉寂了。到了 2010 年 8 月，随着比特币交易所的正式上线，比特币价格开始飞涨，论坛上有人发现了这个帖子，于是回复问“600 美元的比萨好吃吗”（按 2010 年 8 月的比特币价格计算）；2010 年 11 月，有人发出“2600 美元的比萨好吃吗”的跟帖（按 2010 年 11 月的比特币价格计算）。这种跟帖随着比特币价格的飙升越来越多，一直持续到 2016 年 2 月，在原帖子下共有 73 页不同时期的回复。为了纪念这次交易，人们把这笔交易的日子称为“比特币比萨日（Bitcoin Pizza Day）”。

当然，无论是发明比特币的中本聪，还是后来用比特币购买比萨的 Laszlo Hanyecz，他们在当时无论如何也没有想到，不到 10 年的时间，比特币已经成为数字货币领域的翘楚，2017 年 12 月最高达到约 20 000 美元的价格。换言之，如果当时 Laszlo Hanyecz 没有花 10 000 个比特币买比萨，而是一直持有到 2017 年 12 月，那么他的资产将达到 2 亿美元！

TIPS:

2010 年，比特币的创始人中本聪选择了隐退，从网络上消失。在此后很长一段时间内，全球各地先后有不同的人士对外宣称自己就是中本聪，但均未能得到全世界的认可。

1.2.3　比特币的底层技术是区块链

从诞生至今，比特币的价格屡次波动，但总体上处于持续上升的状态。不过，在比特币系统中，最重要的并不是“币”的概念，而是一个没有中心存储机构的“账本”的概念。“比特币”只是在这个账本上使用的记账单位，同时作为价值交易的载体。可以这么说，比特币系统在本质上就是一个基于互联网的去中心化账本，而区块链是这个账本的底层结构。

既然区块链是一个账本，那么这个账本和我们传统的账本有什么不同？我们知道，账本上的内容必须是唯一的，这导致记账天然是中心化的行为。然而，中心化的记账却有一些显而易见的弱点：一旦这个中心出现问题，如被篡改、被损坏，整个系统就会面临信任危机乃至崩溃。

那么，我们能否构建一个去中心化的、不依赖任何第三方的、可信的记账系统呢？

中本聪设计的比特币区块链通过竞争记账的方式解决了去中心化的记账系统的一致性问题。

所谓竞争记账，就是以系统中每个节点（计算机）的计算能力即“算力”来竞争取得记账权的一种机制。在比特币系统中，大约每 10 分钟进行一轮算力竞争（后来被大家形象化地称为“挖矿”），竞争的胜利者就获得一次记账的权利，这样，在一定时间内，只有竞争的胜利者才能记账并向其他节点同步新增账本信息。

那么，在一个去中心化的系统中，谁有权判定算力竞争的胜负呢？比特币系统是通过一个称为“工作量证明（Proof of Work，PoW）”的机制来完成的，即由所有参与节点竞争计算出特定难度的“题目”，最先完成计算并与“答案”要求相符的节点取得胜利，获得记账的权利。

TIPS:

算力竞争是要付出成本的，没有激励，节点就没有进行竞争的动力。在中本聪的设计里，每轮竞争胜出并完成记账的节点将获得系统给予的一定数量的比特币奖励。这个奖励的过程也是比特币的发行过程。

在这个系统中，每个节点只需要根据自身利益行事，比如参与算力竞争的节点只需要“更快、更强”地完成计算，获得记账权，就能获取更多的比特币奖励，这种出于“自利”目的进行的竞争，最终造就了保护系统安全的庞大算力基础。在这种巧妙的安排下，比特币获得了越来越多的信任和越来越高的价值，进而吸引了更多的资源投入其中，成为一个正向循环的经济系统。

1.2.4 区块链全球发展的不均衡特点

随着比特币的发展和壮大，其底层技术——区块链也吸引了越来越多人的注意，人们认识到了区块链技术所蕴含的革命性力量，全球都在积极投入对区块链的研究。

自 2009 年以来，区块链技术经过一段时间的研究和比特币等一些成功应用的积累，在 2015 年爆发，成为全球性的热点。自 2017 年以来，随着比特币价格的暴涨，区块链技术再次成为全球性的资本投入、科技研发的热点，各国的政府、中央银行、金融机构、科技企业及各类投资机构都在强烈关注这一技术的进展。在资本的激发下，各种各样的区块链应用、区块链概念正在层出不穷地涌现。

TIPS:

乐观主义者认为，区块链是继大型机、个人电脑、互联网、移动/社交网络之后的第五次颠覆式创新，是人类信用进化史上继血亲信用、贵金属信用、央行纸币信用之后的第四个里程碑。区块链技术有望彻底重塑人类社会活动形态，并实现从目前的信息互联网向价值互联网的转变。

但是我们应该看到，区块链的发展受到各国的经济发展水平、监管环境、网络基础、信用文化、计算机软硬件水平等外部因素的影响，这也导致在全球各个国家和地区、在各个相关行业里，区块链的发展存在较大的差异。

以区块链的经典应用——比特币为例，从全球范围来看，在部分国家，比特币属于经济生活的热点，资金、人才、技术不断向其集中；而在另一些国家，则根本无人参与。其中，美国、德国、中国在比特币的参与程度上处于全球前 3 位，而非洲、南美洲等区域只有很少的节点参与。比特币可访问节点数量全球前九大国家在总节点数量中所占比例如表 1.1 所示。

表 1.1　比特币可访问节点数量全球前九大国家在总节点数量中所占比例

序　号	国　家	比特币可访问节点数量占总节点数量比例（%）
1	美国	23.67
2	德国	17.62
3	中国	12.41
4	法国	6.35
5	荷兰	4.52
6	加拿大	3.52
7	英国	3.26
8	俄罗斯	3.23
9	新加坡	1.94

1.2.5　区块链在全球的发展现状

客观地说，区块链在全球的快速发展是受区块链典型应用——比特币的快速发展带动的。经过多年的发展，比特币在全球的接受程度越来越高。由于比特币作为数字货币具有一些与现有货币完全不同的特点（比如：去中心化；可在全球自由流通，在任意一台电脑上管理；具有安全性，只能通过专属私钥来控制；匿名、免税、免监管；可直接支付，无中间成本），因而带来了对比特币的各种需求，投机炒作资金也不断介入，其币值在近几年发生了剧烈波动，但总体上一直保持上涨的走势，如图 1.2 所示。从图中可以看到，2017 年年末，比特币价格最高上冲到 20 000 美元。

价格上涨的财富效应吸引了更多的人参与到比特币的挖矿和交易之中。由于用户持有比特币是要存储在比特币钱包（Wallet）中的，因此，比特币钱包的用户数量变化（见图 1.3）可以反映出用户持续增长的实际情况。数据显示，到 2017 年年底，比特币钱包的用户数量超过 2000 万人。

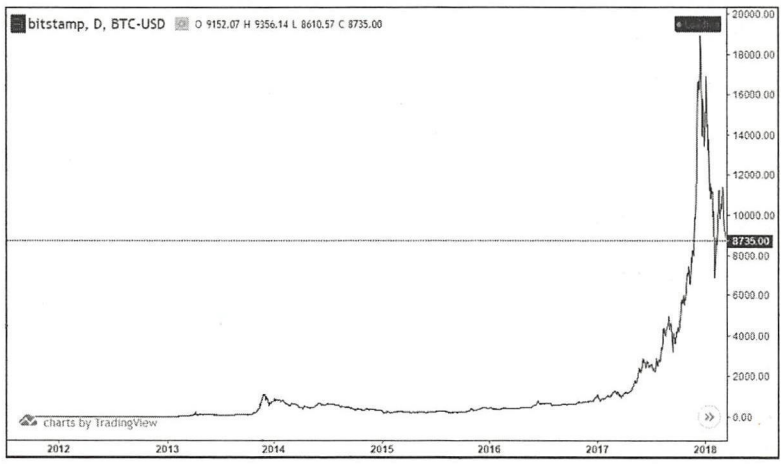


图 1.2　自 2012 年以来比特币价格走势图（单位：美元）

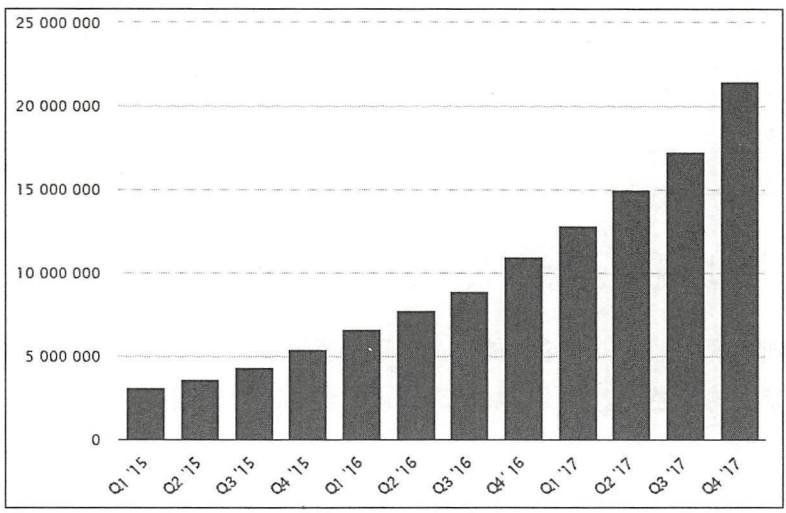


图 1.3　比特币钱包的用户数量（2015—2017 年）

随着币值的上涨及挖矿所带来的比特币数量的持续增多，比特币的市值也在不断增长。截至 2018 年年初，比特币的市值已经达到 1500 亿美元，如图 1.4 所示。

在比特币市值上涨的带动下，其他各类数字货币的价格也出现了类似的走势，如以太币、瑞波币等的价格与最初相比均有了大幅增长，并由此引发了全球范围内发行虚拟货币的投资热潮。如图 1.5 所示为以太币自 2015 年以来的价格走势。

仅供非商业用途或交流学习使用

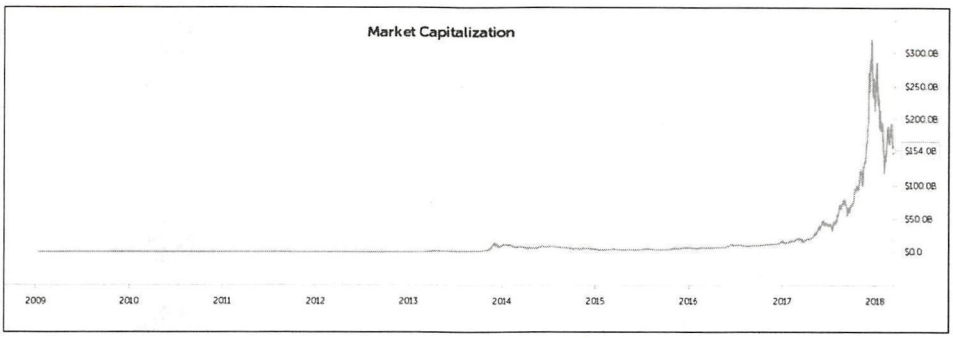


图 1.4　比特币自 2009 年以来市值走势图

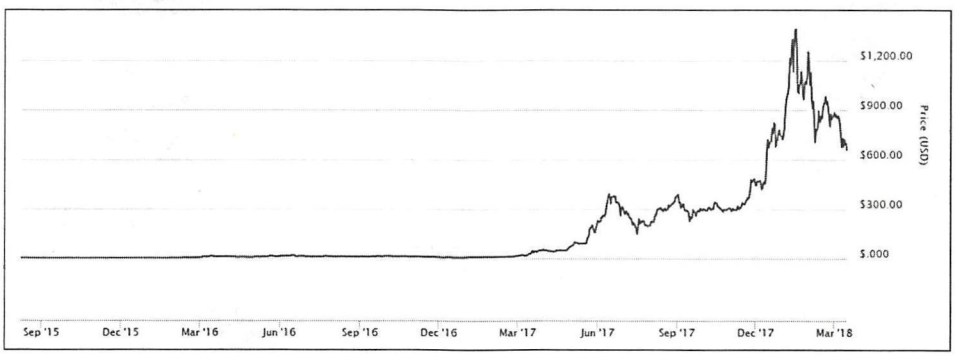


图 1.5　以太币自 2015 年以来的价格走势

由于数字货币的带动，大量的社会资本持续涌入区块链领域，全球对于区块链的研究开发热潮也不断高涨，在区块链技术的各个环节都涌现出了一批公司，致力于推动区块链技术全方位落地。如图 1.6 所示为全球范围内较为领先的各类区块链企业[①]，行业类型包括医疗、审计、支付、公证、物联网（物品的追溯、回溯）、供应链等，区块链的产业环节涵盖支付、程序开发、挖矿、数字货币交易所、媒体平台等各个相关环节。截至 2018 年 4 月 24 日，全球共有 1200 余种数字货币，总市值达近 4000 亿美元，与 2016 年年底仅 177 亿美元的市值相比，增长了 22 倍。

① 图片来源：https://zhuanlan.zhihu.com/p/27034912。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

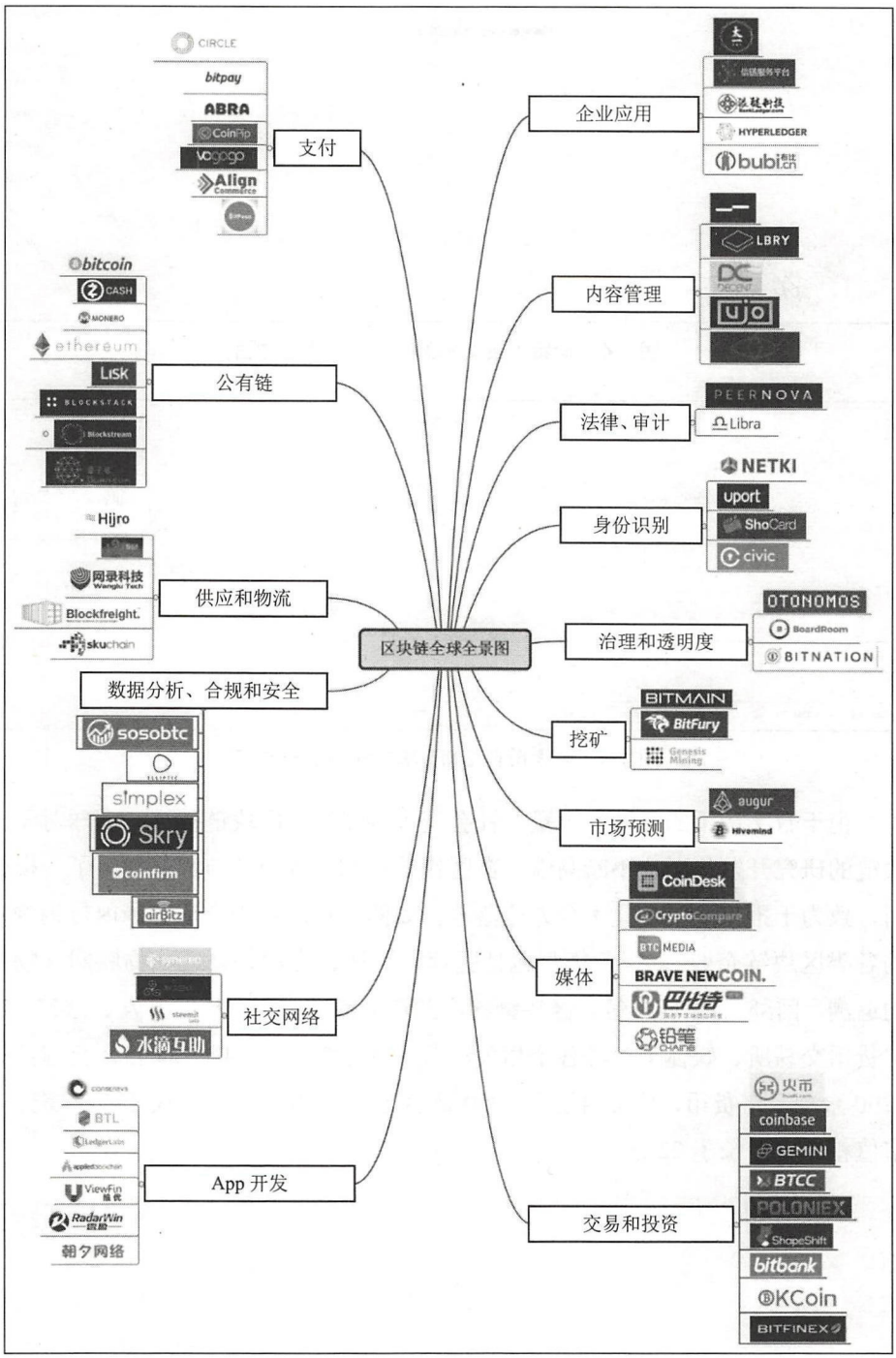


图 1.6　全球范围内较为领先的各类区块链企业

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

据有关机构研究，在全球范围内，由于区块链的应用，可以为银行等机构降低 30%的潜在成本，每年为银行等机构节约成本 80 亿～120 亿美元。各行各业都会受到区块链技术的冲击和变革，比如，消费产品、制造业、科技开发、银行、媒体及电信业等已经有区块链应用落地，健康医疗、生物科技等领域也正在计划引入区块链技术。据德勤预计，全球区块链市场在 2025 年前有望达到 200 亿美元的规模。

TIPS:

受到数字货币价格快速上涨的影响，区块链行业得到了全球资本的热切关注，各类私募基金、投资机构等迅速进入区块链行业，区块链领域的私募股权融资金额快速增长。

一方面，自 2016 年以来，以 ICO（Initial Coin Offerings）形式融资的区块链初创企业快速增长，超过了以传统的私募股权融资等方式融资的区块链初创企业。自 2016 年 1 月至 2017 年第三季度，全球已有超过 250 起 ICO 交易发生，2017 年全年募集资金超过 20 亿美元。

另一方面，在区块链领域，传统的私募股权投资、VC 风险投资等项目募资交易数量继续增长，2017 年共发生 188 起募资交易，比 2016 年增长超过 36%；总募资金额约为 9.45 亿美元，比 2016 年上升 73%。

世界各国政府对于区块链技术也都表示出了积极的态度。爱沙尼亚政府与“e 健康”基金会合作，在 2016 年就发布了区块链医疗数据平台 Guardtime，还在全球率先使用以区块链技术为基础的电子投票服务，使爱沙尼亚公民可以参与美国纳斯达克市场的交易。美国拥有世界最多的数字加密货币用户数量和比特币交易量，硅谷则是大量区块链初创企业的发源地。丹麦对数字货币持开放态度，丹麦央行虽然宣布比特币不是货币，但对于在本国使用比特币却并不禁止，该国也有一批比特币创业企业和交易平台。瑞典金融监管局已经公开宣布比特币可以作为一种支付手段。芬兰宣布对比特币购买免征增值税。总的看来，面对区块链技术变革及相关应用，全球许多国家和地区都积极迎接，而并非拒之门外。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

1.3 区块链在中国的发展现状

2016 年国务院印发的《“十三五”国家信息化规划》首次将区块链列入我国的国家信息化规划，并将其定为战略性前沿技术之一。

2017 年 5 月 16 日，在杭州举行的“区块链技术和应用峰会暨首届中国区块链开发大赛”上，在工业和信息化部信息化和软件服务业司的指导下，中国区块链技术和产业发展论坛公布了《区块链和分布式账本技术参考架构》标准，这是首个在政府指导下的国内区块链基础标准，对区块链产业生态发展具有重要意义。

2018 年 3 月 3 日，中国政府在召开的两会中提出：建立跨地域、跨行业，能够面向整个社会开放的数据共享平台，加强数据安全立法，同时逐步加大引入人工智能和区块链技术，推动大数据与人工智能、区块链等新技术的融合。

中国区块链行业发展的特点如下。

1.3.1 重视底层突破，区块链技术创新加速

2014—2017 年中国及美国区块链领域公开专利数量情况显示，中美两国公司在区块链方面申请的相关专利数量均呈明显上升趋势，其中，美国公司的区块链专利申请数量从 2014 年的 150 件增加到 2017 年前 7 个月的 390 件，中国公司的区块链专利申请数量从 2014 年的 2 件增加到 2017 年前 7 个月的 428 件，中国公司的区块链专利申请数量增速超过美国公司。2018 年年初，全球知识产权产业媒体 IPRdaily 联合 incoPat 创新指数研究中心发布了“2017 全球区块链企业专利排行榜（前 100 名）”，在前 100 名中，中国上榜的企业占比为 49%，美国上榜的企业占比为 33%；中国的阿里巴巴排名第一，Bank of America（美国银行）排名第二。数据再次说明中国企业深度进入区块链技术创新领域。

1.3.2 资本快速进入，区块链融资增长迅猛

2014 年至 2017 年 7 月，全球区块链领域私募投资金额总体呈现增长趋势，其中，美国公司在区块链领域的私募股权融资金额从 2014 年的 2.12 亿美元增

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



长到 2017 年的 3.94 亿美元，增长幅度达 85.85%；而在此期间，中国公司在区块链领域的私募股权融资金额从 0.16 亿美元增长到 0.76 亿美元，增长达 3.75 倍。数据显示，虽然中国公司在区块链领域的私募股权融资总体规模小于美国，但增长速度明显高于美国。

1.3.3　全产业链布局，区块链应用领域逐步拓展

得益于区块链技术的持续创新，区块链在中国的发展也在向各个环节拓展，资本投入的方向比较多元。2014 年至 2017 年 7 月，中国区块链领域私募股权投资的投资方向包括挖矿、钱包、虚拟货币、基础设施、底层技术、交易平台、信息服务、区块链应用 8 个领域，基本包含了区块链产业链的全部组成环节。在所投资的区块链应用类型中，私募股权投资的投资方向又包括数据服务、金融行业应用、认证确权、文化娱乐等 10 个领域，占比较高的领域是数据服务、金融行业应用和认证确权，3 项累计占比达 79%。

1.3.4　抱团发展，各类区块链行业组织纷纷成立

2015 年 12 月至 2017 年年末，中国成立了与区块链相关的行业协会或联盟近 20 个，如中国区块链应用研究中心、GBBC 中国中心、中关村区块链联盟、中国电子学会区块链专委会、中国信通院可信区块链联盟等。这些区块链专业组织为行业参与人员提供了不同的交流及合作平台，对于中国区块链行业的长期、健康发展发挥了有益的作用。

为了促进相关人才的快速成长，中国各高校也纷纷开设多种形式的区块链技术的教育和培训项目，为中国区块链行业创新发展输送人才。比如，清华大学的创客空间 iCenter 举办“区块链技术开发公开课”；同济大学与苏州市合作成立“苏州同济金融科技研究院”，提供区块链技术与应用研发、测评、人才培养及产业孵化等综合性服务；北邮在线区块链教育与研究中心开展区块链实践培训，举办 EMBA 国际总裁班等。其他各种区块链培训也在蓬勃发展，如由中国区块链应用研究中心与GBBC合作组织的区块链应用培训目前已为中国区块链行业培训认证了近千名专业人才。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



1.4 中国区块链行业的发展前景

尽管在发展过程中遇到了一些曲折，但是中国区块链行业前景仍然乐观。因为从全球来看，中国的互联网行业已经发展到了一定的高度，无论是资金、技术、人才还是社会认知，中国人对于互联网基础上的各种创新具备极高的认可度。

区块链技术是在互联网基础上创新、发展出来的新科技，它融合了网络、信用、金融与技术，在中国这片土地上的发展前景和发展速度令世界瞩目。

1.4.1 “90 后”创业者人群大量入场

区块链技术作为一个新兴领域，其发展的基础是互联网，因此，成长在数字社会的“90 后”人群具有天然的优势，他们凭借敏锐的眼光发现了这一蓝海领域，先进入者将占据极大的先发优势。作为互联网时代的创业者，“90 后”人群正在积极地进入这一领域进行创业。

根据互联网金融博物馆（the Museum of FinTech）对金融科技与区块链创新企业的调研，中国有超过 200 家区块链公司的创业者是“90 后”，还有一批“90 后”区块链创业者及投资人正在广泛地参与全球区块链行业的竞争，代表着未来的“90 后”人群与区块链技术的紧密结合，预示着区块链技术具备良好的发展空间。根据移动招聘平台 BOSS 直聘发布的《2018 旺季人才趋势报告》，2018 年年初发布区块链相关岗位人才招聘信息的公司数量同比增长 4.6 倍，说明区块链行业的快速发展对人才的渴求程度大增。

1.4.2 大型企业积极参与，区块链技术基础更加深厚

除大量的创业企业外，基于区块链技术的发展潜力，传统企业、大型国企等其他公司也纷纷宣布涉足这一领域，布局未来的发展空间。

中国平安成为国内首家加入 R3 区块链联盟的机构，目前已在资产交易和征信两个场景中上线了区块链技术；民营企业万向集团专门成立了万向区块链



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



实验室；中国银联与 IBM 合作，开展了“使用区块链技术的跨行积分兑换系统”的研究。

在互联网企业方面，百度与美国支付公司 Circle 达成战略合作，于 2018 年 2 月上线了自己的区块链应用“区块链数字狗”，以实际运营来探索区块链的场景式应用；阿里系的蚂蚁金服将打造基于区块链技术的公益平台，并申领了数十项专利，2017 年 8 月阿里健康与江苏省常州市合作推出首个基于医疗场景的区块链应用“医联体+区块链”试点项目，2017 年 11 月 8 日阿里巴巴集团、蚂蚁金服集团与雄安新区签署了战略合作协议，将承建数字雄安的区块链实施平台；腾讯公司加入了可信区块链联盟，2017 年 4 月，由腾讯 FiT、腾讯研究院、腾讯公共战略委员会办公室、腾讯战略发展部联合编制的腾讯区块链方案白皮书已在腾讯区块链官网（trustsql.qq.com）上正式发布，相关区块链技术已经被腾讯应用在供应链金融、物流信息等多个领域；中食、中粮等传统国企与太一云合作推进中国食品区块链；国家版权局与版权交易中心联盟联合发布了中国版权链；其他众多上市公司、大型金融机构也纷纷推出区块链发展计划。与各种创业企业相比，这些国企、行业巨头积极参与区块链技术的研发，不仅投入了更雄厚的资金和研究团队，也带来了更丰富的、可供区块链技术落地的场景。

1.4.3　全国各地高度支持区块链发展

国务院印发的《“十三五”国家信息化规划》将区块链与大数据、人工智能、机器深度学习等新技术列入国家“十三五”期间的布局重点。

中国人民银行印发了《中国金融业信息技术“十三五”发展规划》，明确提出积极推进区块链、人工智能等新技术的应用研究；2017 年 10 月，工业和信息化部发布了《中国区块链技术和应用发展白皮书》，这是国内首个区块链官方指导文件。

与此同时，中国人民银行正在进行国家数字货币试点，区块链也是其实现的技术之一。2017 年，央行下属的数字货币研究所申请区块链专利 33 件，央



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



行的印制科学技术研究所申请区块链专利 22 件；如果合并计算央行的专利数量（55 件），则可以排到全球第一。

与此同时，全国各地（特别是沿海地区）纷纷成立区块链实验室、研究院，对接企业，促进区块链技术的落地。目前，深圳、杭州、广州、贵阳、赣州等地都在积极建立区块链发展专区，给予区块链技术特别扶植政策。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



chapter TWO

第2章

区块链的商业价值

作为一种具备无限想象空间的革命性技术，区块链目前仍在快速发展之中，全球有无数的资金进入这一领域，各行各业的商业巨头也都在敞开怀抱迎接这一技术。甚至有人开玩笑地将区块链之前的互联网称为“古典互联网”，认为区块链所代表的才是“现代互联网”，与区块链结合的产业才能在未来的发展中赢得先机。

由此可见，区块链领域将迎来商业化的全面发展。作为底层的技术，区块链的商业价值将在不同的层面与不同的行业结合，从而得以体现。

2.1 区块链在银行业的应用

当前，全世界的银行都是“中心化”的，因为银行是作为资金流通的信任机构而存在的：人们把钱存放到银行里，进行资金的交易；不同的银行之间通过结算中心进行交易的验证、确认和记账，最后实现清算。银行必须建立庞大、安全的信息系统，一方面存储所有客户的账户及交易信息，另一方面与其他银行、金融机构进行信息交换，从而实现交易和资金的传输及交叉验证。

在这个过程中，交易变得无比烦琐，因为信任的缺失，所以每个环节都需要不断地确认和对账。比如做一笔跨境资金交易，资金到账的过程需要数天，而且面临着很高的交易费用，在时间、金钱等方面增加了交易成本，但是这些

仅供非商业用途或交流学习使用

过程在目前的银行体系环境下是无法去掉的，是人们为了获得信任、推进交易必须付出的成本。

而且，一旦银行的信息系统被黑客攻陷，入侵者很容易篡改系统，将巨额资金转走，比如 2017 年 11 月，尼泊尔的 NIC 亚洲银行被黑客通过计算机系统攻击，被盗资金高达 4.6 亿尼泊尔卢比（约合 4700 万元人民币）。另外，在银行的交易中，如果一方违约或拒不履行协议，就会导致另一方交易失败而损失惨重。全球性管理咨询公司奥利弗·怀曼（Oliver Wyman）发布的报告显示，每年金融行业用来弥补交易失败的费用高达 800 亿～6500 亿美元。这也是行业的巨大“痛点”。

2.1.1 区块链对银行业的改变

面对传统银行业所存在的种种不足，区块链技术展示了巨大的商业价值，因为它可以解决银行业所面临的这些“痛点”问题。

首先，区块链天然具有防篡改的特点。由于区块链是通过分布式网络在全球成千上万个节点上进行共同记账的，没有中心化的账本，所以，即使有黑客通过网络攻击某些节点中的账本并进行篡改，他也无法同一时间改动全网络超过 51%的节点中的账本内容，这使得其篡改行为毫无意义，因为这样的篡改很容易被废弃、不被区块链网络所认可。

其次，由于共识信任机制的存在，任何一笔资金交易在区块链上的流动过程都会得到全网的确认和认可，从而不再需要银行这种中心化机构进行增信，交易可以在点对点的交易双方之间直接发生，从而彻底降低交易成本、提高交易效率。区块链还可以使用计算机算法自动处理和结算交易，而不需要第三方验证，也不需要通过中间人进行身份验证，支付的过程更快、更安全、更容易审计。如无意外，传统银行的后台清算方式都将被区块链彻底颠覆。

2.1.2 国内外银行业的区块链应用

目前国内外银行业已经进入实际研发的区块链商业开发有以下几类。

1. 民生银行：加入 R3 区块链联盟，搭建区块链云平台

民生银行于 2016 年 11 月宣布加入 R3 区块链联盟。民生银行信息科技部有关负责人表示，民生银行加入 R3 区块链联盟的目的是寻求与国际大型金融机构的合作机会，学习并探索区块链分布式账簿技术的业务模式。除此之外，民生银行也搭建了区块链云平台，并且对区块链共识算法、智能合约、交易记账、数据传输、智能钱包、去中心化应用等进行深入研究。

2. 招商银行：直连清算系统正式应用

招商银行主要将区块链技术应用到直连清算系统，这是招商银行内部用于跨境清算的系统（仅用于该公司内部，属于私有链）。招商银行有 5 个海外机构（1 家子行 4 家分行）：永隆银行、新加坡分行、伦敦分行、卢森堡分行、纽约分行。以往的清算系统只能支持分行与总行之间进行清算。随着直连清算系统的应用，在跨境清算场景下，区块链去中心化的系统使得分行之间也可以发起清算请求，同时私有链封闭的网络环境保证了交易的安全性，最终使得招商银行内部跨境清算的速度达到秒级。

3. 中国邮政储蓄银行：利用区块链开展资产托管业务

2017 年 1 月 10 日，中国邮政储蓄银行（以下简称“邮储银行”）与 IBM 公司共同召开新闻发布会表示，邮储银行在资产托管业务场景中利用区块链技术实现了中间环节的缩减、交易成本的降低及风险管理水平的提高。邮储银行的这个系统于 2016 年 11 月上线，在真实的业务环境中已经顺利执行了上百笔交易。

传统资产托管业务涉及资产委托方、资产管理方、资产托管方及投资顾问等多方金融机构，各方都有自己的信息系统，传统的交易主要通过电话、传真、邮件等方式进行信用校验，而区块链正好解决了相互信用校验的成本，业务环节缩短了 60%～80%。

4. 浙商银行：上线移动数字汇票平台、应收款链平台

2017 年 1 月 3 日，浙商银行发布首个基于区块链的移动数字汇票平台，为企业与个人提供在移动客户端签发、签收、转让、买卖、兑付移动数字汇票的功能。除了可以降低多方参与部门的成本，采用区块链技术的最大优点是以数字资产的方式进行汇票存储、交易，且不易丢失、无法篡改。从纸质汇票、电

子汇票再到基于区块链技术的移动汇票，实现了层级跨越。

2017 年 8 月，浙商银行运用区块链技术打造了应收款链平台，可以帮助企业盘活经营中形成的大量应收款。该平台采用区块链、智能合约等创新技术，与浙商银行自身设计开发的企银合作平台结合，专门用于办理企业应收款的签发、承兑、保兑、支付、转让、质押、兑付等业务。通过应收款链平台，企业可将报表中沉淀的应收款改造为高效、安全的电子支付结算和融资工具，随时对外支付和融资。

浙商银行此次推出的企业应收款链平台主要采用了区块链的去中心化、分布式账本和智能合约等重要特性。

（1）去中心化：实现了企业的唯一签名。在区块链上，密钥一经生成便不能更改，类似于企业身份证，企业在签名后完成应收款记账，银行等任何第三方均无法篡改应收款交易信息，最大限度地保证应收款信息安全。

（2）分布式账本：采用区块链分布式账本技术记录应收款信息，彻底改变传统应收款依赖纸质或电子数据可靠性的限制，从技术上排除了数据被篡改、被伪造的各种可能，从而使其记录的应收款信息成为高可信的凭据。

（3）智能合约：区块链通过智能合约技术，可保证应收款各类交易按照约定的规则完成，改变传统依赖人工操作的应收款履约处理，改由区块链系统根据智能合约规则自动、无条件履约。截至 2017 年年末，应收款链平台上线短短 4 个月时间，已签约企业达 100 余家，业务累计发生额达近 30 亿元，实现融资余额达 25 亿元。

5. IBM 公司的超级账本（Hyperledger）区块链项目

除上述银行业代表积极参与区块链应用以外，在银行业以外的科技公司也在积极投入区块链在银行业的商业开发，比如 IBM 公司投资的 Hyperledger Fabric 项目。Hyperledger Fabric 是 IBM 研发的一个贸易融资平台，目前已经发布了 1.0 版本，其目的是利用区块链改进银行国际收支方面的业务。Hyperledger Fabric 是在由 Linux 基金会开发的开源项目 Hyperledger 基础上进一步开发出来的。与当前流行的其他区块链项目不同的是，在 Hyperledger Fabric 中是不需要加密货币的，进入网络需要得到授权，获得访问权限（具有一定的私密性，而

不是完全公开的），其开发语言为 Golang 和 Java，交易类型既可以是匿名的也可以是公开的。

Hyperledger Fabric 通过 IBM Cloud 运行，运用区块链技术可以使特定交易中所有银行之间达成安全互连。通过区块链，该项目允许多个参与者融入整个金融供应链流程，从而实现前所未有的交易透明度。目前，包括德意志银行、汇丰银行、比利时联合银行、Natixis 银行、荷兰合作银行、法国兴业银行和意大利联合信贷银行在内的大银行都是 Hyperledger Fabric 的重要支持者。

2.2　区块链与电子商务

区块链对电子商务的改变必将是颠覆性的。在区块链技术出现之前，电子商务是互联网应用于日常生活、替代传统商业的典范。我国电子商务快速发展，将人流、物流、资金流、信息流“四流合一”，整体业务发展迅猛，已经初步形成功能完善的市场体系，形成了以淘宝、京东为代表的多个大型电子商务平台；与此同时，跨境电子交易发展迅速，海淘、海外代购等形式的电商平台也大量出现，带动了全球商品的交流。

当前，我国电商的基本格局已经形成，成为拉动国民经济增长的重要动力，但是伴随其快速发展也带来了许多问题。①信用机制亟待完善。②假货问题难以杜绝。无论是商家还是各大电商平台，都无法建立一个有效的商品防伪机制，尽管电商平台推出了很多打击假冒伪劣商品的措施，但依然无法杜绝假冒伪劣商品，消费者权益无法得到有效保障。③交易数据存在安全隐患。在用户使用电商平台的过程中，其个人隐私数据问题逐渐暴露出来。电商平台怎样保护用户数据不被窃取、怎样控制电商平台有限度地收集利用客户访问数据进行推送，目前尚未能够取得有效的成果。

可以说，随着电子商务的快速发展，消费者在享受便利性的同时，已经越来越多地感受到了电子商务固有的缺陷所带来的隐患，那就是“中心化的结构”。以淘宝为例，它是一个集中的平台，消费者要通过这个平台与商户连接，支付的货款要通过支付宝来为交易双方增加信用。在互联网、电子商务发展初期，

支付宝是一个无与伦比的创举，因为它用第三方机构的信用来为互联网上陌生的交易双方提供信用保证，从而推动了交易的进行。但是它同时增加了交易的流程，提高了交易的成本。在海量用户的支持下，支付宝成为除银行外的资金集聚平台，反过来成了交易的中心。

此外，为了保证用户的信息安全和支付安全，支付宝自身也必须投入大量的财力，以保证交易的快捷、准确、安全。例如每年的“双十一”，阿里巴巴都像经历一场“信息战争”，从服务器、网络带宽、IT 人员支持等各方面做好保障，使海量用户的网购行为得以快捷、顺利完成。这其中巨额的投入看起来与用户无关，但实际上来自从用户一侧获取的盈利。可以说，离开了淘宝、京东这样的电子商务平台，在目前的电商环境下，交易无法顺利地进行。

TIPS:

区块链的出现必将改变这一现象，使交易真正成为买卖双方的直接行为，无须第三方的中介桥梁作用。

首先，用于交易的商品，从生产者开始，其每种原材料都是有据可查、可以追溯、不可更改的，因为原材料的来源、产地等数据已经作为基础信息写入区块中，并链接到整个数据链上，得到了所有人的共识。从这个角度来说，制假者根本无法造假，因为原材料的造假将导致该件商品不可追溯、不能得到区块链网络的共识，造假的商品无法进入区块链，根本就不能销售出去。

其次是隐私信息的泄露。当前电商平台与大数据的所谓结合带来了客户隐私信息的泄露问题，一些电子商务的服务商通过购买、采集等多种手段收集了客户访问电商平台的数据，再有针对性地加以推送，提高客单价，形成了“价格歧视”；一些非法机构则通过黑客等手段窃取电商用户的个人信息、账号、密码等情况，然后转让牟利，用户个人信息的泄露导致了垃圾短信、垃圾邮件的泛滥成灾，而由于目前网络的特点，这些行为无法被追踪和追责，泄密、窃密行为不断发生。

这些情况与当前“中心化”的电商架构有着密切的联系，因为用户的个人信息、偏好等都要输入中心化的电商平台，而这个中心化的电商平台却缺乏监督，仅靠自己的良知来指导和控制其使用客户数据的行为。

TIPS:

一旦引入区块链技术，商户与用户之间直接连接，双方的交易一方面采用匿名机制，由非对称加密技术进行身份验证，从而使商户收集用户个人信息的行为无法实施；另一方面，由于交易的数据需要得到全网的公认，并链接到区块链上，如果有某个环节泄露了用户个人信息，那么通过区块链的数据回溯，可以准确定位泄密者，从而使泄密者面临法律、共识等多方面的制裁，非法行为必将销声匿迹。

最后是支付的安全性与便利性。当前电子商务的主要支付过程大多借助支付宝等第三方支付平台进行交易资金的支付，这是阿里巴巴在信用不足的客观环境下创造性的解决方案，为电子商务的发展打下了坚实的基础。可以说，支付宝实际上创立了一家民间的“银行”，它为交易的双方进行了信用的验证，提供了支付的保障，减少了交易双方的风险。

但是从本质上来讲，支付宝是一个“中心化”的第三方机构，它在本来可以直接支付的交易双方之间插入了一个环节，实际上还是增加了支付交易的成本；只不过在当前的电商环境下，支付宝作为中间环节显然是必不可少的，只有当区块链技术真正应用之后，才能让支付宝彻底“退休”。

TIPS:

在区块链技术下，交易双方的支付直接进行，每笔支付都会得到全网节点的共同确认，一旦完成确认，进入区块链，则无法篡改。在这种情况下，交易双方无须借助支付宝即可实现安全、可信的资金转让，支付宝的存在价值也就消失了。

2.3　区块链在法律行业的应用

区块链的首要特点就是通过分布式网络的全部节点的共识建立起了无法篡改的账本，记录了网络内的所有交易，使得交易变得可信、可靠，这与法律行业的需求有着最大程度的吻合，因此，区块链的出现给法律行业带来了无与伦比的商业价值。

2.3.1 区块链证据

区块链在法律行业的价值首先体现在“无法篡改”特性的应用上，也就是对于法律诉讼等所需的证据资料的记录和保存，使证据资料得到目前最为安全的保管。2017 年 8 月 16 日，全球首个电子存证区块链联盟——大规模商用电子存证区块链联盟“法链”宣告成立。法链是由 Onchain、微软（中国）及法大大等多个机构参与建立和运营的证据记录和保存系统，是一个多方参与的开放式区块链联盟。

法链由上海法链网络科技有限公司负责日常运营和管理，技术架构则基于 Onchain 自主开源的 DNA 而搭建。据介绍，目前每天有超过 80 万份来自法大大的电子合同、邮件等文件在法链上存证。法链中的参与机构各司其职，打造开放平台。在联盟中，Onchain 提供的底层区块链技术与法大大电子合同“深度融合”，每份电子合同的签署时间、签署主体、文件哈希值等数字指纹信息将会被广播到法链所有成员的各个存储节点上，并将文件哈希值信息同步存储在国家权威电子数据司法鉴定中心。

另外，我国自 2013 年 1 月 1 日起正式施行的新修订的《民事诉讼法》明确规定，电子数据也可作为证据。但在具体实施中，电子证据因为易修改、难以记录完整等特点导致公信力不足。利用区块链技术来建立和运营电子证据记录和保存系统，意味着电子证据一经存储，任何一方都无法篡改，并且电子证据会存储在每个参与者处，帮助扫清电子数据成为有效的司法证据的障碍。

2.3.2 智能合约

区块链在法律行业的价值其次体现在智能合约的应用上。目前在重大合约的签署过程中，法律顾问、律师事务所的服务是不可或缺的，通过律师的工作为合约双方提供各方面合规信息的查询，确保双方具备履约能力和条件，保证合约得到履行，最终通过双方对于律师的信任延伸到对于交易对手的信任，促成交易的进展。在这个过程中，律师的存在所起的作用为增加信任，为此律师付出了时间成本、精力成本和财务成本，交易双方则需要在交易费用之外额外付出聘请律师的费用，同时交易的实施时间也大幅延长，效率大为降低。

区块链技术的应用将彻底改变这一现象。通过应用智能合约，交易双方的信任由区块链的共识机制进行保障；同时由于合约条件的存在，履约条件一经触发则合约自动履行，履行后也将得到整个区块链网络的确认，交易双方无法自行改动。例如，在区块链上建立一份向各个股东按某个金额进行分红的智能合约，当达到约定的利润数额时，分红自动执行，或者在约定的时间即可进行。由此可见，在智能合约的交易方式下，不再需要律师等法律中介机构，必将提高交易效率、降低交易成本。

2.3.3　区块链权证

区块链在法律行业的价值最后体现在对交易的认证上。目前对于一些法律文件、交易文件等，由于交易各方涉及利益相关无法体现公允，因此一般会借助外部的公证机构进行再次认定，以第三方机构的公信力来保障交易资料、权证的可靠性。最主要的原因就是在现有的商业环境中，法律文件、交易文件、产权证明等各类与交易密切相关的文书权证等易被伪造，而且受制于交易各方的各自立场，对权证的可靠性证明不容易被认可，导致交易各方必须向外部机构“借”来信任。

如果区块链技术得以应用到交易文件、资信证明、权证等方面，当这些文书权证的全部来源及过程都能够进入区块中，以加密形式保存下来，并通过前后链接形成完整的区块链，得到全部网络节点的共识后，那么其可信度、可靠性等多方面都会得到各方的绝对认可，交易各方不再需要借助公证机构的第三方增信作用，交易的时间成本、财务成本也就随之降低。

区块链的应用还将改变产权记录这一长期由各类中心化机构管理的法律要素。时至今日，类似于房产、土地产权、矿权等不动产的产权登记和管理，以及汽车所有权等动产的产权登记和管理，仍然主要依托各类产权管理机构进行，而且采取的方式和方法以“中心化”的管理为主，实物证书采用数字化的形式存储在产权管理机构的数据库里，而这个数据库是单一的，最多在少量的服务器上有一些备份。

由此带来几个方面的低效：一是数据库的安全性受制于管理机构在硬件和软件上的投入水平，一旦出现了问题，数据库受损后可能会导致信息丢失，此

时所能依靠的是实物证书，效率低下；二是当需要确认产权时，只能访问管理机构的数据库，由所有的节点向这个中心机构发送数据请求，为了保证产权数据库的安全、可靠，管理机构必然会设计出对访问请求的控制措施，因此降低了各类交易者对于产权信息确认的效率。

TIPS:

如果采用了区块链技术，那么产权的形成过程，比如一座房屋从购置各类建筑材料到由开发商建成装潢漂亮的房屋，已经在各个节点上形成了交易记录，建成后的产权记录进一步由区块链系统的各个节点进行确认，同时这条产权区块链是在各个节点上同步存放的，随时可以访问、追溯，产权记录无法更改，某个节点的数据损毁并不影响整体账本的数据安全。一旦达到这样的境界，则产权管理、产权确认无须再经由中心化机构的通道，交易双方随时可以进行。

2.4 区块链在影音娱乐和媒体行业的应用

伴随着经济的快速发展，中国的影视娱乐、音乐和媒体行业也进入了快速发展通道。最近几年借助互联网的快速传播和推动，影音娱乐和媒体行业再次井喷，造就了大量的明星艺人，电影市场日趋火爆，行业欣欣向荣。

但是，在行业快速发展的环境下，也有一些需要改进的地方，比如，对于行业来说，原创内容日趋稀缺，版权和成本等各种压力持续增加；同时，视频、音乐的盗版现象始终无法根治，令版权拥有者的收益大量流失，间接影响了行业的健康发展。另外，媒体行业在互联网的冲击下发生了巨变，传统纸媒日渐衰落，借助网络的自媒体蓬勃兴起，一方面带来了信息的海量冲击，另一方面导致了信息的复制泛滥，优质内容创作者的利益需要由平台来变现，降低了整个媒体行业的运营效率。

以电影行业为例，当制作方完成一部电影的制作后，需要通过发行方送达全球各大影院上线放映，传递到千千万万的观众眼前；或者通过视频网站等大型的传播机构，借助互联网来传递到观众这一端。

TIPS:

这是一个很长的链条，中间存在的各个环节一方面为创作者提供了将内容传递到观众的通道，另一方面切分了一部分来自海量观众提供的收益。如果没有这样的中间层级，那么创作者的收益会更高，观众也能够以更低的付出欣赏到电影内容。

但是由于没有让创作者与观众直接对接的技术和机制，所以全世界只能以这样低效的方式进行电影的制作与发行。在这种情况下应用区块链技术，将形成强大的价值体现。

从创作者的角度来说，他的作品一旦完成，进入区块链的数据记录中，并获得分布式网络的所有节点的共识之后，这份版权是无法被篡改的，这意味着无须其他的机构来进行版权的保护。

从观众的角度来说，当他进入区块链，购买某部电影用以自己观赏时，交易会由区块链进行记录，影片内容将直接来自创作者的节点，减少中间的发行方、电影院等通道，降低延时、节约成本；更重要的是，创作者方面通过在电影上附加权限控制，可以使观众侧无法通过区块链进行复制、传播，因为这样的复制和传播缺少前序区块链的支持，将不能获得其他节点的共识，在区块链网络下将无法通过网络自由传播，从而形成了对盗版现象的根本杜绝。

另外，区块链技术的应用也将提高创作者的利益。在目前的电影行业制作模式下，绝大部分电影由“中心化”的大型电影制作公司来制作完成，为了保证票房，这些电影公司一般只会乐于拍摄由明星演员出演的电影，其他的电影很难得到拍摄的机会；电影公司拍完后，又通过大型的发行公司控制了影院这一渠道，从而使小众的电影很难有机会被看到。而在区块链下，由于创作者直接面对分布式网络的各个节点，只要有人愿意观看小众、另类的电影，通过区块链完成交易，那么电影制作人将直接通过区块链交易获得观众给予的资金，实现自己的收益，避免了中心化的发行公司不制作此类电影的弊端，电影制作人可以自行制作完成这类的电影，满足个性化的需求。

TIPS:

通过区块链，可以使少量明星之外的大批创作者凭借个性化的创作实现自身的收益。甚至通过在区块链上建立智能合约，一首歌曲的创作者可以约定好观众播放的付费金额或比例，每次歌曲的播放将直接带给版权所有者收益，进一步提高了交易效率。

从长远来看，区块链技术对于影视行业的影响是颠覆性的，因为区块链技术能很好地解决与信用、发行、版权管理和盗版有关的问题，随着不同的应用逐渐出现，必将给影音娱乐、媒体等行业带来全新的格局。比如，已经运营数年的视频网站“拍片网”，运用区块链技术给每段视频加上数字证书，任何交易记录都被记入区块链存证，通过区块链的溯源可以很容易识别盗版的视频；另外，拍片网也正在建立全球影视作品版权交易网络，允许任何影视、传媒、出版从业者上传影视作品进行版权认证，并在全球范围内进行版权交易，上传内容“包括但不限于创意构思、剧本、视频素材、音乐素材、三维模型、制作工程、电影、电视剧集、纪录片、艺术短片等”。具体的流程包括：

（1）将视频等数字媒体内容在分布式网络中建立数字资产的身份。

（2）制定收益分配明确的智能合约，并发布到区块链中，受到共识机制的确认，合约无法被篡改，也就无法形成盗版。

（3）由区块链网络中去中心化的节点之间进行点对点交易，视频内容在创作者和用户之间直接交易。

2.5 区块链在医疗行业的应用

区块链在发展过程中逐步形成了区块链 1.0、区块链 2.0 和区块链 3.0 的概念。区块链 1.0 就是以比特币为代表的各类虚拟货币。区块链 2.0 是指区块链技术在其他金融领域的运用。区块链 3.0 则意味着将区块链应用的领域扩展到金融行业之外，进入人类生活的各个方面，因为区块链能够通过全部网络节点共识实现信息的自证明，不再依靠中心化的权威机构来获得信任或建立信用，通过增加信任来实现信息共享，提高整个系统的运转效率。医疗健康领域是区块链技术发挥价值的另一个重要领域。

在现实生活中，由于医院等机构仍然沿用的是中心化数据库或文件柜式管理医疗数据资料（如患者档案），所以容易出现网络操作错误或者黑客攻击等问题，患者私密信息泄露情况时有发生，比如，Anthem 曾经泄露 8000 万病人和雇员的记录，UCLA Health 曾经泄露 450 万病人的数据。

随着医学的发展，以及不久的将来指纹数据应用和基因数据检测手段的普及，更多的生物信息将会进入个人的医疗数据中，一旦发生泄露，将会导致灾难性的后果。

TIPS:

因此，医疗数据急需一种安全的、不可篡改的方式进行保管。在这方面，区块链的价值无法替代。医疗机构正通过区块链技术实现对患者隐私信息的记录、保管及合理利用（主要是在疾病控制和治疗过程中）。

在医疗方面可以探索的区块链应用方向有如下几个。

1. 电子健康病例

将个人的医疗记录保存到区块链上，不再由各家医院集中保存。从患者的角度来说，他可以充分获得自己的医疗记录和病史情况。各个节点的记录一旦进入区块链，就成为无法改动的信息。其他医院在对患者进行诊断时，通过患者的授权可以在区块链上访问其记录，未获授权者无法访问，这样就可以控制患者隐私信息的泄露。

患者在投保时，保险公司也无须再担心不能掌握全面的健康数据。通过访问电子健康病例的区块链，保险公司可以不断溯源，充分获得患者的信息。但同样不能向外泄密，因为区块链的验证机制使非授权访问信息无法进入区块，也就不能通过验证。

2. 药品防伪

通过药品的数字编码在区块链上溯源，可以了解药品的生产全过程，从而确定是否为假冒产品。

3. 健康信息共享

区块链技术的出现令原来存储在不同中心化机构或者散落在个人节点上的

医疗数据及健康信息有机会链接起来，形成新的模式。

通过打破不同医院间的信息壁垒，不同节点的医院都可以访问去中心化的区块链数据库，不同医疗机构患者的诊疗过程和过程数据都可以在有追溯的情况下被医生访问，减少网络延迟，提高交流效率，能够实现多网络节点的数据流和可信数据分享，使得医疗数据在安全保护下产生新的价值。

另外，随着可穿戴设备的广泛应用，为跟踪个人健康状况而收集的大量数据不断传输至云端和智能手机中，但目前还只能用于个人或社交层面的信息交流，并没有传递到医生手里，用作治疗时的参考，产生更多的辅助医疗价值。

TIPS:

利用区块链从家用设备、可穿戴设备或家庭保健提供者处获取患者个人提供或传输的治疗效果、日常生理体征数据等相关信息，是拓宽不良事件安全性监测范围的一种途径。

在医疗技术开发、新型药物研制等其他更加强调数据保护特性的医疗环境中，区块链的价值也更加突显。通过将临床试验所有历史记录输入区块链，可以确保实时记录试验方案，使得试验结果来源可查、去向可控，真实性有保证。基于区块链的技术架构，这类数据能够得到很好的保护，有需要时也可以在研发机构之间以智能合约的方式进行分享和交流。

2.6 结语

前文我们列举分析了区块链技术在几个典型行业的商业价值，从技术本质上来看，区块链无疑将带给相关行业巨大的改变和进步，带来效率的提升和社会的进步。目前绝大多数专家认为，区块链技术将对当前几乎所有的行业产生颠覆性的影响。虽然我们在此无法列举出每个行业应用区块链技术的商业价值，但是，由于区块链具有“从无信任处创造信任”的独特魅力，而信任是一切创造价值活动的基石，所以我们可以毫不犹豫地说：区块链将最大化地创造出商业价值。

chapter THREE

第3章

区块链技术原理

区块链技术的出现与人类历史上带来第一次工业革命的蒸汽机技术的出现极为相似，它们都是直接以产品（或应用）的方式进入人类社会，并向全社会的其他行业扩散的。区块链技术的出现是先有比特币的应用，然后才引发了对其底层技术——区块链的开发热潮。

迄今为止，比特币是区块链在全球范围内参与范围极广、接受程度极高的公有链之一。比特币的出现是在IT行业持续不断地对数字货币、分布式数据及去中心化信用机制的成果集成基础上，开创性地建立了一套确保区块链系统能够稳定和可持续运行的机制，奠定了区块链技术发展的基础。了解和研究区块链技术，最好的方式就是从比特币着手。

3.1 比特币带来了区块链

比特币不是凭空出现的。在它出现之前，广大的程序员、技术极客已经进行了很长时间的摸索。正如我们在第1章中所介绍的，在比特币出现之前，在邮件讨论组“密码朋克”中由成员讨论、发明或失败的数字货币和支付系统多达数十个。

TIPS:

英国密码学家亚当·贝克（Adam Back）在 1997 年发明了哈希现金（Hash Cash），其中用到了工作量证明（Proof of Work，PoW）机制。这个机制本来是用于解决互联网垃圾信息问题的，后来被引入比特币中，成为比特币的核心要素之一。

哈伯和斯托尼塔（Haber and Stornetta）在 1997 年提出了一个用时间戳的方法保证数字文件安全的协议，这个协议成为比特币区块链协议的原型。

戴伟（Wei Dai）是一位兴趣广泛的密码学专家，他在 1998 年发明了电子货币 B-money，强调点对点的交易和不可更改的交易记录，这一点也被比特币借鉴。

在前人研究的基础上，2008 年 10 月 31 日纽约时间下午 2 点 10 分，一个自称中本聪的人在密码朋克的电邮系统中发布了白皮书 *Bitcoin: A Peer-to-Peer Electronic Cash System*（《比特币：一种点对点的电子现金系统》），描述了一个新的货币体系，这就是比特币（Bitcoin）。同年 11 月 16 日，中本聪提供了比特币代码的初期版本。

2009 年 1 月 3 日，中本聪在位于芬兰赫尔辛基的一台小型服务器上挖出了比特币的第一个区块——创世区块（Genesis Block），并获得了“首矿”奖励——50 个比特币。

从创世区块至今，比特币在最初数年处于沉寂状态，直到 2011 年 5 月，由于参与人数逐渐增加，其价格产生了一轮上涨，达到近 20 美元，但随后下跌到 2 美元；从 2013 年开始，比特币价格逐步上涨，吸引了越来越多的人进入，最终在 2017 年年底触及 20 000 美元的历史高点，成为数字货币的典型代表，并带动了数千种其他类型的数字货币的出现。目前比特币的总市值达到 150 亿美元左右，挖矿形成的比特币总量超过 1600 万个，比特币的区块数量已经超过 50 万个。

比特币是区块链技术在全球范围内的第一个成熟的应用，至今保持着旺盛的生命力，它标志着区块链技术从理念进入实践，它持续增长的价值效应不断吸引全球的人才、资金进入区块链领域，促进了区块链技术的进一步发展。比特币的出现和发展历程证明了区块链技术的生命力，以及对于商业、金融等社会形态产生巨大影响的能力。

3.2　比特币白皮书

中本聪发表的比特币白皮书第一次详细说明了比特币的运行机制。只有深入了解了这份白皮书，我们才能对比特币有一个深刻的理解。随着研究区块链的人士增多，在网络上已经可以看到比较好的中文翻译，为我们的研究减少了障碍。以下为比特币白皮书的中文译文[①]。

3.2.1　简介

互联网上的贸易几乎都需要借助金融机构作为可资信赖的第三方来处理电子支付信息。虽然这类系统在绝大多数情况下都运作良好，但是仍然内生性地受制于“基于信用的模式（Trust Based Model）”的弱点。我们无法实现完全不可逆的交易，因为金融机构总是不可避免地会出面协调争端。而金融中介的存在也会增加交易的成本，并且限制了实际可行的最小交易规模，也限制了日常的小额支付交易。

并且潜在的损失还在于，很多商品和服务本身是无法退货的，如果缺乏不可逆的支付手段，互联网的贸易就会大大受限。因为有潜在的退款的可能，就需要交易双方拥有信任。而商家也必须提防自己的客户，因此会向客户索取完全不必要的个人信息。而在实际的商业行为中，一定比例的欺诈性客户也被认为是不可避免的，相关损失视作销售费用处理。而在使用物理现金的情况下，这些销售费用和支付问题上的不确定性却是可以避免的，因为此时没有第三方信用中介的存在。

所以，我们非常需要这样一种电子支付系统，它基于密码学原理而不基于信用，使得任何达成一致的双方能够直接进行支付，从而不需要第三方中介的参与。杜绝回滚（Reverse）支付交易的可能，这就可以保护特定的卖家免于欺诈；而对于想要保护买家的人来说，在此环境下设立通常的第三方担保机制也可谓轻松愉快。

①参考网址：https://github.com/GammaGao/bitcoinwhitepaper/blob/master/bitcoin_cn.pdf。

在这篇论文中，我们将提出一种通过点对点分布式的时间戳服务器来生成依照时间前后排列并加以记录的电子交易证明，从而解决双重支付问题。只要诚实的节点所控制的计算能力的总和大于有合作关系的（Cooperating）攻击者的计算能力的总和，该系统就是安全的。

3.2.2 交易

我们定义一枚电子货币（an Electronic Coin）是这样的一串数字签名：每位所有者通过对前一次交易和下一位所有者的公钥（Public Key）签署一个随机散列的数字签名，并将这个签名附加在这枚电子货币的末尾，电子货币就发送给了下一位所有者。而收款人通过对签名进行检验，就能够验证该链条的所有者。如图 3.1 所示为比特币的交易模型。

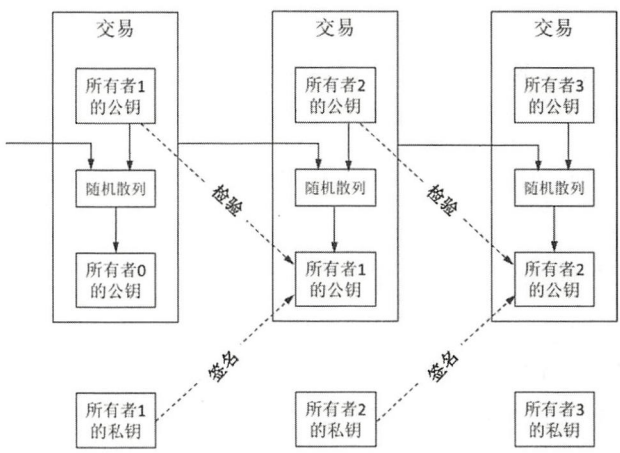


图 3.1 比特币的交易模型

该过程的问题在于，收款人将难以检验之前的某位所有者是否对这枚电子货币进行了双重支付。通常的解决方案就是引入信得过的第三方权威或者类似于造币厂（Mint）的机构来对每笔交易进行检验，以防止双重支付。在每笔交易结束后，这枚电子货币就要被造币厂回收，而造币厂将发行一枚新的电子货币；只有造币厂直接发行的电子货币才算作有效，这样就能够防止双重支付。可是该解决方案的问题在于，整个货币系统的命运完全依赖运作造币厂的公司，因为每笔交易都要经过该造币厂的确认，而该造币厂就好比一家银行。

我们需要收款人有某种方法能够确保之前的所有者没有对更早发生的交易实施签名。从逻辑上来看，为了达到目的，实际上我们需要关注的只是在本次交易之前发生的交易，而不需要关注在这笔交易发生之后是否会有双重支付的尝试。

为了确保某次交易是不存在的，唯一的方法就是获悉之前发生过的所有交易。在造币厂模型里面，造币厂获悉所有的交易，并且决定了交易完成的先后顺序。如果想要在电子系统中排除第三方中介机构，那么交易信息就应当被公开宣布（Publicly Announced），我们需要整个系统内的所有参与者都有唯一公认的历史交易序列。收款人需要确保在交易期间绝大多数的节点都认同该交易是首次出现的。

3.2.3　时间戳服务器

本解决方案首先提出一个“时间戳服务器”的概念。时间戳服务器通过对以区块（Block）形式存在的一组数据实施随机散列而加上时间戳，并将该随机散列进行广播，就像在新闻或世界性新闻组网络（Usenet）中的发帖一样。显然，该时间戳能够证实特定数据必然于某特定时间是的确存在的，因为只有在该时刻存在了才能获取相应的随机散列值。每个时间戳应该将前一个时间戳纳入其随机散列值中，每个随后的时间戳都对之前的一个时间戳进行增强（Reinforcing），这样就形成了一个链条（Chain）。如图 3.2 所示为时间戳示意图。

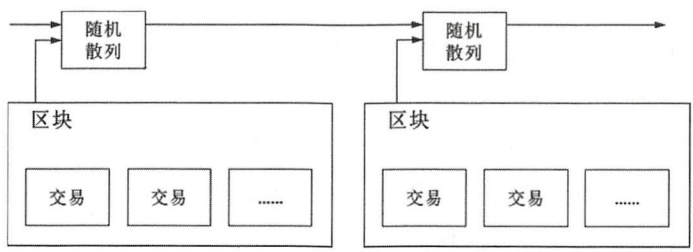


图 3.2　时间戳示意图

3.2.4　工作量证明

为了在点对点的基础上构建一组分散化的时间戳服务器，仅仅像报纸或世

界性新闻组网络一样工作是不够的，我们还需要一个类似于亚当•贝克（Adam Back）提出的哈希现金（Hash Cash）。在进行随机散列运算时，工作量证明机制引入了对某个特定值的扫描工作，比如在 SHA256 算法下，随机散列值以一个或多个 0 开始。那么随着 0 的数目的上升，找到这个解所需要的工作量将呈指数增长，而对结果进行检验则仅需要一次随机散列运算。

我们在区块中补增一个随机数（Nonce），这个随机数要使得该给定区块的随机散列值出现了所需的那么多个 0。我们通过反复尝试来找到这个随机数，直到找到为止，这样我们就构建了一个工作量证明机制。只要该 CPU 耗费的工作量能够满足该工作量证明机制，那么除非重新完成相当的工作量，否则该区块的信息就不可更改。由于之后的区块是链接在该区块之后的，所以想要更改该区块中的信息，还需要重新完成之后所有区块的全部工作量。工作量证明机制的示意图如图 3.3 所示。

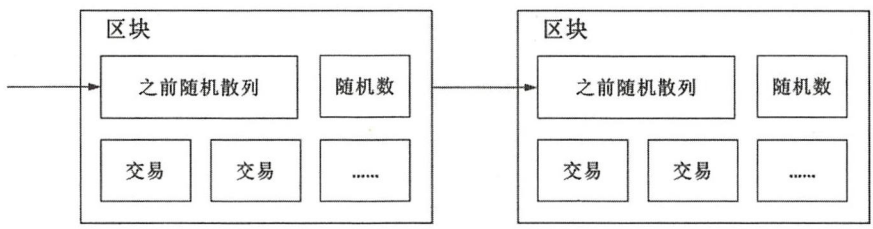


图 3.3　工作量证明机制的示意图

同时，该工作量证明机制还解决了在集体投票表决时谁是大多数的问题。如果决定大多数的方式是基于 IP 地址的，一个 IP 地址一票，那么倘若有人拥有分配大量 IP 地址的权利，该机制就被破坏了。而工作量证明机制的本质则是一个 CPU 一票。“大多数”的决定表达为最长的链，因为最长的链包含了最大的工作量。如果大多数的 CPU 被诚实的节点所控制，那么诚实的链条将以最快的速度延长，并超越其他的竞争链条。如果想要对业已出现的区块进行修改，那么攻击者必须重新完成该区块的工作量外加该区块之后所有区块的工作量，并最终赶上和超越诚实节点的工作量。我们将在后文证明，设想一个较慢的攻击者试图赶上随后的区块，那么其成功概率将呈指数化递减。

另一个问题是，硬件的运算速度在高速增长，而节点参与网络的程度则会有所起伏。为了解决这个问题，工作量证明的难度（the proof-of-work difficulty）

将采用移动平均目标的方法来确定，即令难度指向令每小时生成区块的速度为某个预设的平均数。如果区块生成的速度过快，难度就会提高。

3.2.5　网络

运行该网络的步骤如下：

（1）新的交易向全网进行广播。

（2）每个节点都将收到的交易信息纳入一个区块中。

（3）每个节点都尝试在自己的区块中找到一个具有足够难度的工作量证明。

（4）当一个节点找到了一个工作量证明后，它就向全网进行广播。

（5）当且仅当包含在该区块中的所有交易都是有效的且之前未存在过的，其他节点才认同该区块的有效性。

（6）其他节点表示它们接受该区块，而表示接受的方法则是在该区块的末尾制造新的区块以延长该链条，而将被接受区块的随机散列值视为先于新区块的随机散列值。

节点始终都将最长的链条视为正确的链条，并持续工作和延长它。如果有两个节点同时广播不同版本的新区块，那么其他节点在接收到该区块的时间上将存在先后差别。当此情形，它们将在率先收到的区块基础上进行工作，但也会保留另一条链条，以防后者变成最长的链条。该僵局（Tie）的打破要等到下一个工作量证明被发现，而其中的一条链条被证实为较长的一条，那么在另一条分支链条上工作的节点将转换阵营，开始在较长的链条上工作。

所谓“新的交易要广播”，实际上不需要抵达全部的节点。只要交易信息能够抵达足够多的节点，那么它们将很快被整合进一个区块中。而区块的广播对被丢弃的信息是具有容错能力的。如果一个节点没有收到某特定区块，那么该节点将会发现自己缺失了某个区块，也就可以提出自己下载该区块的请求。

3.2.6 激励

我们约定如此：对每个区块的第一笔交易进行特殊化处理，该交易产生一枚由该区块创造者拥有的新的电子货币。这样就增加了节点支持该网络的激励，并在没有中央集权机构发行货币的情况下，提供了一种将电子货币分配到流通领域的方法。这种将一定数量的新货币持续增加到货币系统中的方法非常类似于耗费资源去挖掘金矿并将黄金注入流通领域。此时，CPU 的时间和电力消耗就是消耗的资源。

另一个激励的来源则是交易费用（Transaction Fees）。如果某笔交易的输出值小于输入值，那么差额就是交易费用，该交易费用将被增加到该区块的激励中。只要既定数量的电子货币已经进入流通领域，那么激励机制就可以逐渐转换为完全依靠交易费用，本货币系统就能够免于通货膨胀。

激励系统也有助于鼓励节点保持诚实。如果有一个贪婪的攻击者能够调集比所有诚实节点加起来还要多的 CPU 算力，那么他就面临一个选择：要么将其用于诚实工作产生新的电子货币，要么将其用于进行二次支付攻击。那么他就会发现，按照规则行事、诚实工作是更有利可图的。因为该等规则使得他能够拥有更多的电子货币，而不是破坏这个系统，使得其自身财富的有效性受损。

3.2.7 回收硬盘空间

如果最近的交易已经被纳入了足够多的区块中，就可以丢弃该交易之前的数据，以回收硬盘空间。为了同时确保不损害区块的随机散列值，交易信息在被随机散列时，被构建成一种默克树（Merkle Tree）的形态，使得只有根（Root）被纳入了区块的随机散列值。通过将该树（Tree）的分支拔除（Stubbing）的方法，老区块就能被压缩。而内部的随机散列值是不必保存的。如图 3.4 所示为回收硬盘空间的过程。

不含交易信息的区块头（Block Header）大小仅有 80 字节。如果我们设定区块生成的速度为每 10 分钟一个，那么每年产生的数据量为 4.2MB（80 Bytes×6×24×365=4.2MB）。2008 年，PC 系统通常的内存容量为 2GB，按照摩尔定律的预言，即使将全部的区块头存储于内存之中都不是问题。



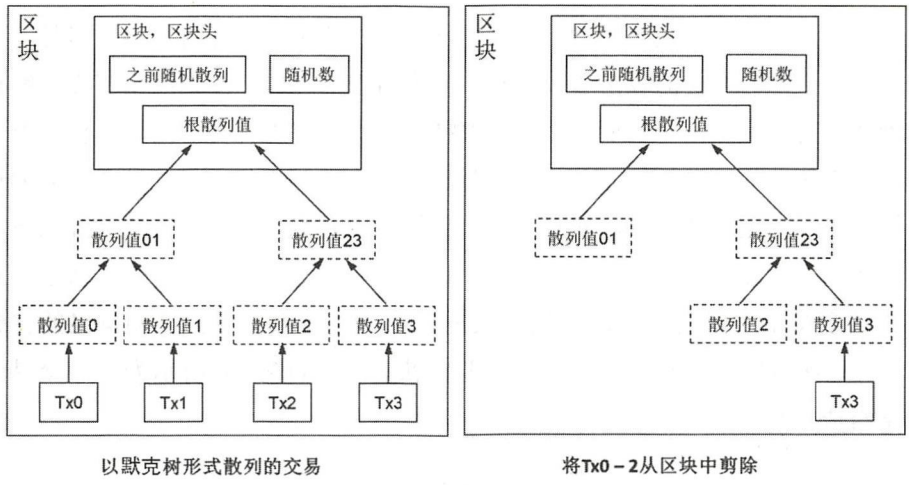


图 3.4　回收硬盘空间的过程

3.2.8　简化的支付确认

在不运行完整网络节点的情况下，也能够对支付进行检验。一个用户需要保留最长的工作量证明链条的区块头的拷贝，它可以不断地向网络发起询问，直到它确信自己拥有最长的链条，并能够通过默克树的分支通向它被加上时间戳并纳入区块的那次交易。节点想要自行检验该交易的有效性原本是不可能的，但通过追溯到链条的某个位置，它就能看到某个节点曾经接受了它，并且于其后追加的区块也进一步证明全网曾经接受了它。如图 3.5 所示为长链检验机制在支付确认上的应用。

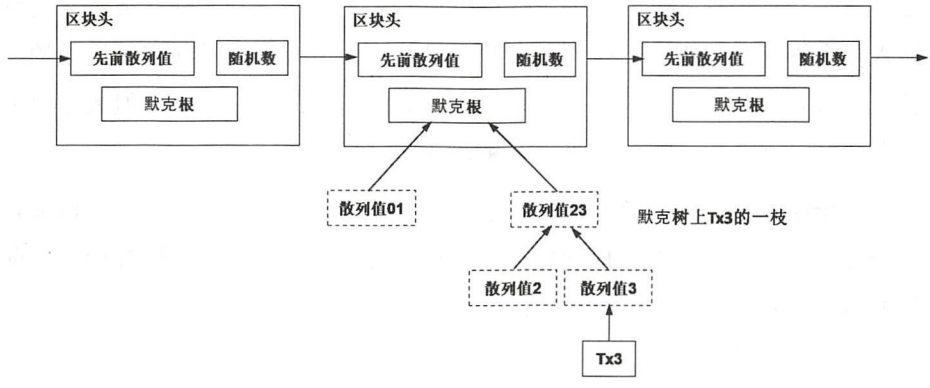


图 3.5　长链校验机制在支付确认上的应用



当此情形，只要诚实的节点控制了网络，检验机制就是可靠的。但是，当全网被一个算力占优的攻击者攻击时，将变得较为脆弱。因为网络节点能够自行确认交易的有效性，只要攻击者能够持续地保持算力优势，简化的机制就会被攻击者伪造的（Fabricated）交易所欺骗。那么一种可行的策略就是，只要他们发现了一个无效的区块，就立刻发出警报，收到警报的用户将立刻开始下载被警告有问题的区块或交易的完整信息，以便对信息的不一致进行判定。对于日常会发生大量收付的商业机构来说，可能仍会希望运行它们自己的完整节点，以保持较大的独立完全性和检验的快速性。如图 3.6 所示为一次交易的完整组成信息。

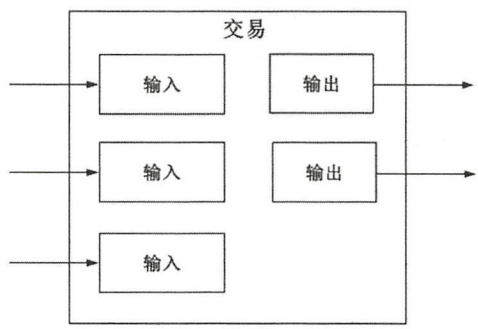


图 3.6　一次交易的完整组成信息

3.2.9　价值的组合与分割

虽然可以单个地对电子货币进行处理，但是对于每个电子货币单独发起一次交易将是一种笨拙的办法。为了使得价值易于组合与分割，交易被设计为可以纳入多个输入和输出。一般而言，输入是某次价值较大的前次交易构成的单一输入，或者由某几个价值较小的前次交易共同构成的并行输入，但是输出最多只有两个：一个用于支付；另一个用于找零（如有）。

需要指出的是，当一笔交易依赖之前的多笔交易时，这些交易又各自依赖多笔交易，但这并不存在任何问题。因为这个工作机制并不需要展开检验之前发生的所有交易历史。



3.2.10　隐私

传统的造币厂模型为交易的参与者提供了一定程度的隐私保护，因为试图向可信任的第三方索取交易信息是严格受限的。但是如果将交易信息向全网进行广播，就意味着这样的方法失效了。但是隐私依然可以得到保护：将公钥保持为匿名。公众得知的信息仅仅是有某个人将一定数量的货币发送给了另一个人，但是难以将该交易同特定的人联系在一起，也就是说，公众难以确定这些人究竟是谁。这同股票交易所发布的信息是类似的，股票交易发生的时间、交易量是记录在案且可供查询的，但是交易双方的身份信息却不予透露。如图 3.7 所示对比了两种隐私模型的特点。

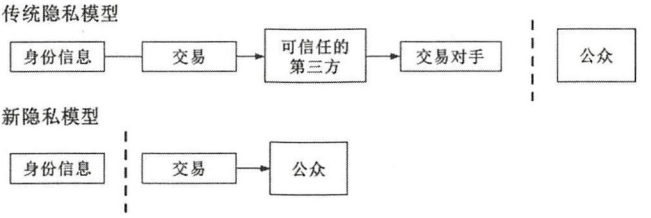


图 3.7　两种隐私模型的特点对比

作为额外的预防措施，使用者可以让每次交易都生成一个新的地址，以确保这些交易不被追溯到一个共同的所有者。

TIPS:

由于并行输入的存在，一定程度上的追溯还是不可避免的，因为并行输入表明这些货币属于同一个所有者。此时的风险在于，如果某个人的某个公钥被确认属于他，就可以追溯出此人的其他很多交易，对其个人隐私带来一定的影响。

3.2.11　计算

设想如下场景：一个攻击者试图比诚实节点产生链条更快地制造替代性区块链。即便它达到了这一目的，但是整个系统也并非就此完全受制于攻击者的独断意志，比如凭空创造价值，或者掠夺本不属于攻击者的货币。这是因为节

仅供非商业用途或交流学习使用

点将不会接受无效的交易，而诚实的节点永远不会接受一个包含了无效信息的区块。一个攻击者能做的最多是更改他自己的交易信息，并试图拿回他刚刚付给别人的钱。

诚实链条和攻击者链条之间的竞赛可以用二叉树随机漫步（Binomial Random Walk）来描述。成功事件被定义为诚实链条延长了一个区块，使其领先性+1；而失败事件则是攻击者的链条被延长了一个区块，使得差距-1。

攻击者成功填补某一既定差距的可能性可以近似地看作赌徒破产问题（Gambler's Ruin Problem）。假定一个赌徒拥有无限的透支信用，然后开始进行潜在次数为无穷的赌博，试图填补上自己的亏空。那么我们可以计算他填补上亏空的概率，也就是该攻击者赶上诚实链条的概率，如下所示：

$$p = \text{诚实节点制造出下一个节点的概率}$$

$$q = \text{攻击者制造出下一个节点的概率}$$

$$q_z = \text{攻击者最终消弭了}z\text{个区块的落后差距}$$

$$q_z = \begin{cases} 1, & p \leqslant q \\ (\frac{q}{p})^2, & p > q \end{cases}$$

假定 $p>q$，那么攻击成功的概率就因为区块数的增长而呈现指数化下降。由于概率是攻击者的敌人，如果他不能幸运且快速地获得成功，那么他获得成功的机会随着时间的流逝就变得愈发渺茫。那么我们考虑一个收款人需要等待多长时间才能足够确信付款人已经难以更改交易了。我们假设付款人是一个支付攻击者，希望让收款人在一段时间内相信他已经付过款了，然后立即将支付的款项重新支付给自己。虽然收款人届时会发现这一点，但为时已晚。

收款人生成了一对新的密钥组合，然后只预留一个较短的时间将公钥发送给付款人。这将可以防止以下情况：付款人预先准备好一个区块链，然后持续地对此区块链进行运算，直到运气让他的区块链超越了诚实链条，方才立即执行支付。当此情形，交易一旦发出，攻击者就开始秘密地准备一条包含了该交易替代版本的平行链条。

然后收款人将等待交易出现在首个区块中，其后链接了 z 个区块。此时，作为诚实区块，他仍然不能确切地知道攻击者已经进展了多少个区块，但是假



仅供非商业用途或交流学习使用

设诚实区块将耗费平均预期时间以产生一个区块，那么攻击者的潜在进展就是一个泊松分布，分布的期望值为：

$$\lambda = z\frac{q}{p}$$

当此情形，为了计算攻击者追赶上的概率，我们将攻击者取得进展区块数量的泊松分布的概率密度乘以在该数量下攻击者依然能够追赶上的概率。

$$\sum_{k=0}^{\infty}\frac{\lambda^k e^{-\lambda}}{k!}\cdot\begin{cases}\left(\frac{q}{p}\right)^{(z-k)} & ,k\leqslant z\\ 1 & ,k>z\end{cases}$$

化为如下形式，避免对无限数列求和：

$$1-\sum_{k=0}^{z}\frac{\lambda^k e^{-\lambda}}{k!}\cdot\left(1-\left(\frac{q}{p}\right)^{(z-k)}\right)$$

写为如下 C 语言代码：

```
    #include double AttackerSuccessProbability(double q, int z)
{
double p = 1.0 - q;
double lambda = z * (q / p);
double sum = 1.0;
int i, k;
for (k = 0; k <= z; k++)
{
double poisson = exp(-lambda);
for (i = 1; i <= k; i++)
poisson *= lambda / i;
sum -= poisson * (1 - pow(q / p, z - k));
}
return sum;
}
```

对其进行运算，我们可以得到如下的概率结果，发现概率对 z 值呈指数化下降。

当 q=0.1 时：

```
z=0 P=1.0000000
z=1 P=0.2045873
z=2 P=0.0509779
z=3 P=0.0131722
z=4 P=0.0034552
```



仅供非商业用途或交流学习使用

```
z=5 P=0.0009137
z=6 P=0.0002428
z=7 P=0.0000647
z=8 P=0.0000173
z=9 P=0.0000046
z=10 P=0.0000012
```

当 q=0.3 时：

```
z=0 P=1.0000000
z=5 P=0.1773523
z=10 P=0.0416605
z=15 P=0.0101008
z=20 P=0.0024804
z=25 P=0.0006132
z=30 P=0.0001522
z=35 P=0.0000379
z=40 P=0.0000095
z=45 P=0.0000024
z=50 P=0.0000006
```

求解令 P<0.1%的 z 值。

为使 P<0.001，则：

```
q=0.10 z=5
q=0.15 z=8
q=0.20 z=11
q=0.25 z=15
q=0.30 z=24
q=0.35 z=41
q=0.40 z=89
q=0.45 z=340
```

3.2.12 结论

我们在此提出了一种不需要信用中介的电子支付系统。我们首先讨论了通常的电子货币的电子签名原理，虽然这种系统为所有权提供了强有力的控制，但是不足以防止双重支付。为了解决这个问题，我们又提出了一种采用工作量证明机制的点对点网络来记录交易的公开信息，只要诚实的节点能够控制绝大多数的 CPU 算力，就能使得攻击者在事实上难以改变交易记录。该网络的强健之处在于它结构上的简洁性。



仅供非商业用途或交流学习使用

节点之间的工作大部分是彼此独立的，只需要很少的协同。每个节点都不需要明确自己的身份，由于交易信息的流动路径并无任何要求，所以只需尽其最大努力传播即可。节点可以随时离开网络，而想重新加入网络也非常容易，只需补充接收离开期间的工作量证明链条即可。节点通过自己的 CPU 算力进行投票，表决它们对有效区块的确认，它们不断延长有效的区块链来表达自己的确认，并拒绝在无效的区块之后延长区块以表示拒绝。本框架包含了构建一个 P2P 电子货币系统所需的全部规则和激励措施。

3.3　比特币系统的参与者

中本聪的比特币白皮书介绍了比特币的一些主要组成元素，当然，由于只有 9 页的篇幅，他对这些组成元素都是点到为止的，只讲明了基本规则。实际上，比特币系统从根本上来说是一个网络协议，规定了比特币的生成规则、交易规则、数据交换格式等。作为一个网络协议，它只规定使用这个协议的用户的行为。系统中的任何节点只要在行为上满足这个协议的要求，就可以被网络接受，而这些行为可以由任何代码实现，并不需要某个特定的代码，这体现了比特币开放的特点，至今它的源代码都是公开在编程社区 GitHub 上的，任何人都可以复制然后改编，开发出自己的数字货币。当然，这也导致了后来数以千计的山寨数字货币在全球出现。

比特币首创性地建立了一整套规则，使得系统得以有效运作。在这套规则下，比特币系统的参与者分为两类：记账节点和交易方。

记账节点的功能就是公开记账、竞争上岗。我们说过，比特币及其底层技术——区块链其实是一个分布式账本。

TIPS:

> 作为账本，就需要记账人员。在一个中心化的账本中，记账人员实质上掌握了权力，因为他们决定了账本的内容。

比特币系统作为分布式、去中心化的账本，为了避免产生中心化的记账人员，采用了全网多点记账的竞争模式，即：对于每 10 分钟在全网络内发生的交



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

易，理论上每个节点都可以进行记录，但记录完成后到底采纳哪个节点的账务记录，需要由这些记账节点进行竞争，并通过全网共识确认。也就是说，每个节点都可以计算由系统设计的同一套算法规则下的“谜题”（这个“谜题”与交易记录相关），最先算出结果的节点可以向全网公布，经过全网节点验证确定无误后，其他节点就以这个记录为起点，进入新一轮的记账权竞争。

要理解这一过程，我们可以参考中国传统的“打麻将”游戏（虽然参与的人数少了一点）。

在打麻将时，4 个参与者是平等的，没有哪个参与者有特权；游戏开始后，打麻将的人要取出 13 张牌，这类似于比特币节点对 10 分钟的交易进行封装的过程；牌取好后，打麻将的人进行组合、吃碰、调整牌面等一系列操作，力争在 4 人之中最快完成组合，最快达到和牌规则（如清一色等）要求，这一过程类似于比特币节点的算力计算，争取最快计算出系统认可的算法结果；某个打麻将的人率先和牌，于是推倒向大家公示自己已经和牌了，这类似于某个比特币节点完成计算后率先向全网“推牌”公示自己的答案；打麻将的另外 3 人对推牌人的和牌验证无误后认同他赢牌，并向其支付各自的下注，4 人重新洗牌开始下一轮打牌，这类似于比特币节点对公示答案的节点进行确认后，以这个记录为起点，进入下一轮的记账权竞争。

打麻将和牌的人会赢得一定的金额（虽然这笔钱是其他人下的注），而在比特币系统中，完成记录的节点将给自己的地址增加一定数量（目前是 12.5 个）的比特币，相当于得到了奖励；如果这个记录得到了全网公认，则它的记录也就成为全网共识，它就“拥有”了 12.5 个比特币，可以用于转让、买卖、兑换等交易过程，而且这笔比特币来自记账时的第一笔交易（在比特币中称为币基交易），今后的每个记录区块都可以追溯至此，并且经全网确认，合乎共识规则。因此，即使在没有中心化机构认证的情况下，在比特币网络中，这个节点既创立了这 12.5 个比特币，也得到了这些比特币的所有权。

TIPS:

在比特币网络中，从事记账的节点被称为“矿工”，形象地表示了它们从比特币网络中“挖”出比特币的过程。



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

从上述过程中可以看出，比特币是设计者用来对维护系统交易的矿工给出的激励措施。因为矿工进行记账是付出了电力、算力等资源的，如果没有收益，就没有人愿意进行记账操作，比特币系统的区块就不能形成，最终丧失了存在的根基。

另外，中本聪设计的规则是比特币奖励每 4 年减半，目前已经从最初的 50 个比特币/区块减少到 12.5 个比特币/区块，未来将趋近于 0，最终产生的比特币总量为 2100 万个。当所有的比特币“挖”出之后，矿工的收益来自交易者提供的交易费用，按目前每 10 分钟 1 个区块的速度，这个时间预计是 2140 年。

比特币作为一种数字货币，它的价值体现为可以作为各种交易的价值交换媒介。在比特币系统中存在大量的交易，比如最初有人用 10 000 个比特币购买两个比萨的行为。在比特币系统中，交易分为两类：第一类交易是特殊交易，即产生比特币的交易，由系统默认记录在每个区块的第一笔；第二类交易是正常的交易，比如 A 向 B 转让一定数量的比特币等，这些交易则由矿工记录后封装到一个又一个区块之中，互相链接，构成区块链。比特币是一本分布式的、由全网络每个参与节点共同维护的“总账”，这本“总账”的页数在随时随地地增加，因为它的每一“页”就是一个区块（Block），随着系统中的交易量不断增长，区块也在不断地增加和链接上去，“总账”的页数也在不断增加。

TIPS:

“总账”的每一“页”都有一个唯一的“页码”，换言之，每个区块都有一个自己的 ID，这个 ID 是对包含了前一个区块的 ID 的数据进行哈希计算得来的。正是由于这一机制，每个区块都可以通过 ID 链接（Chain）起来，这也是区块链（Blockchain）名称的由来。

矿工在挖出了比特币后，需要通过交易获取其他利益。而本身并不是矿工的交易者因为没有通过挖矿取得比特币，如果想参与到比特币的交易中来，就需要使用现实生活中的货币（如美元、欧元等）向持有比特币的人以一定的比例购买比特币，从而使比特币这一数字货币具有了现实货币的定价。随着参与者数量不断增加，对比特币的需求也在持续增长，从而导致了 2013 年在全球范围内对比特币的爆炒。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

比特币的底层技术正是区块链，它的运行机制也就是区块链的运行机制。通过了解比特币的运行机制，我们就能理解区块链的运行机制。

3.4 比特币区块

比特币本身就是区块链，这条链上的每个区块是怎样构造出来的？简单来说，它是由矿工节点“挖”出来的。通过矿工的挖矿，在比特币上的交易得到了确认，同时也以去中心化的共识模式发行了新的比特币，这个过程的结果就是产生一个又一个的区块（Block）。

3.4.1 比特币的交易过程

任何一个数字货币系统，以及区块链系统，其存在的目的都是实现交易。在比特币系统中，同样存在着各种各样的交易，每笔交易都包括一些数据，如支付方的比特币钱包 ID、支付的比特币数额、接收方的比特币钱包 ID、交易的详情，以及交易的费用等。在每笔交易发生后，由节点向比特币的全部网络进行广播，等待得到确认。只有交易被记入区块，才表示该交易得到了全网的确认。目前一个区块的产生时间约为 10 分钟，因此这笔交易和 10 分钟内的其他交易一起，由矿工节点进行组装。

每笔交易包括两个方向：输入（Input）和输出（Output）。二者的数量和比特币金额可以不完全一致，但只能输入略大于输出，输出不能大于输入。当输入略大于输出时，多出的比特币金额就是支付给矿工的交易费用。输入和输出的对象都是比特币钱包地址。输入的比特币数量都是有来源的、可以被追溯的。如图 3.8 所示是比特币交易示意图。

与现实生活中类似，是否支付交易费用会影响矿工处理交易的优先级，包含足够费用的交易会更快速地被打包进下一个区块中；反之，交易费用不足或者没有交易费用的交易可能会被推迟，当然，矿工会在尽职基础上在若干个区块之后处理这些交易（因为即使确认全部免费的交易，矿工还是会得到系统的奖励，所以他们还是会处理无交易费用的交易的），在极端情况下这些交易可能根本不会被处理。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

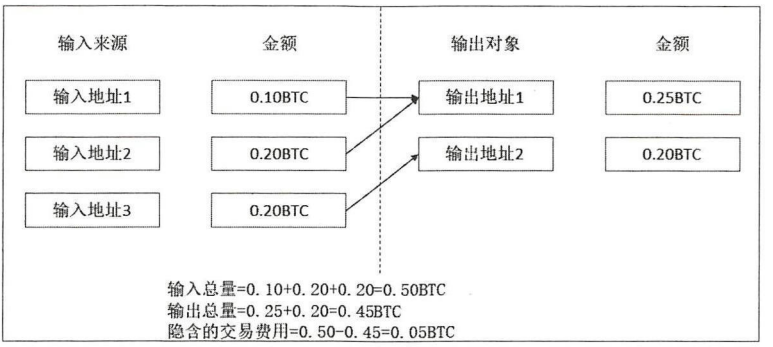


图 3.8　比特币交易示意图

TIPS:

所以，尽管交易费用不是强制的，而且零交易费用的交易最终也可能会被处理，但是，为了提高被处理的优先级，参与者将越来越多地选择支付交易费用。

随着时间的推移，交易费用也在发生变化。刚开始时交易费用是固定比例的，但随着网络容量和交易量的增加，自 2016 年年初以来，比特币网络容量的限制已经造成交易之间的竞争，从而导致人们支付更高的费用以获得尽快处理，免费交易彻底消失，非常低费用的交易很少被处理，有时甚至不会在网络上传播。

3.4.2　比特币挖矿

除上述与生活中的商业交易行为相类似的普通支付交易外，每个矿工在组装区块时，首先装入的交易是给自己奖励额定数量（当前为 12.5 个）比特币的交易，称为币基（Coinbase）交易。它的输入端是没有钱包地址的，而显示 No Inputs (Newly Generated Coins)，也就是“最新产生的比特币”；输出端是矿工节点自己的钱包地址。

如果这个区块得到了所有节点的确认，就意味着全网都承认这位矿工的钱包里增加了 12.5 个比特币，这 12.5 个比特币似乎是“从天而降”的，没有输入来源（钱包地址），因此它可以被看作一种奖励，可由矿工自由支配，全网都认可。这就是比特币的发行机制，没有中央银行，没有美联储，但必须在得到全网认可后才能取得新增比特币的所有权，因此称之为“去中心化”的数字货币发行机制。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

矿工要得到这笔奖励，就需要做出实实在在的贡献，也就是对当前这一时段的所有交易进行确认。这是一项需要算力的过程，普通参与者在算力的硬件装置、电力消耗上不具备竞争优势，在竞争过程中最终会由一些组织或专业矿工取得胜利。具体的过程是这样的：

矿工节点在接入网络后，保持对全网络的监听，实时接收由钱包软件广播的交易信息。每个收到交易的比特币节点将会首先验证该交易，确保只有合乎规则的交易才会在网络中广播，不合规则的无效交易在第一个节点处就被废弃。

在当前时点上，矿工已经通过网络获知了自前一区块完成以来一定时间段内每笔交易的信息，然后使用哈希算法 SHA256 计算每笔交易的哈希值。当计算完成后，再把这些交易两两组合，再次计算 SHA256 哈希值，并不断重复这一过程，形成类似于金字塔的结构，最终得到一个所有交易的 SHA256 哈希值，称为“默克根（Merkle Root）”。

TIPS:

在取得默克根后，矿工将其与前一区块的 ID、一个随机数再次组合，进行 SHA256 哈希计算，得到一个哈希值。将这个哈希值与给定的条件进行对照。通常情况下给定的条件是：要求哈希值的前 20 位为 0。这是一个相当苛刻的条件，达成的概率在百万分之一的水平上。

因此，矿工的第一次随机哈希显然不能匹配给定条件的要求，于是增加随机值，再次进行哈希计算和结果比对。这个过程一般要持续几百万次，从概率上来说已经接近于买彩票中奖的概率了，而且同一时点上全网络内在不同地理位置的矿工都在进行计算竞争，所以对于硬件的算力要求较高，还要消耗大量的电力。如果某个矿工在最短的时间内第一个完成了“前 20 位为 0”这一给定条件的哈希计算，那么他就可以向全网发送通知，声明他已经完成了一个区块的构造，其他节点根据该矿工发出的通知信息（包括前一区块的 ID、交易默克根、随机数等），经计算确认后确实能够哈希得到“前 20 位为 0”，大家就公认他创建的区块是成立的，他的币基交易也是成立的，于是大家都认可了他可以拥有 12.5 个全新的比特币。而其他矿工则马上以他构造的这个区块作为父区块，即用这个区块的 ID 作为哈希计算的参数，进行下一个区块的计算竞争。整个

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

过程的规则非常透明、一致，就看谁的算力强、运气好，在所有参与的矿工中尽早计算出符合给定条件的最终哈希值，就能够最终拥有新的比特币。

当然，对新区块的确认也是有“程序”的。每个矿工节点在收到新区块的通知后，会按照以下几个步骤对新区块进行验证，验证通过之后才认可，验证不通过的区块会被丢弃。

（1）区块的数据结构语法成立。

（2）区块头的哈希值小于目标难度（这是最重要的标准，说明“解出了谜题”）。

（3）区块时间戳早于验证时刻未来 2 小时。

（4）区块大小在长度限制之内。

（5）第一笔交易只能是币基交易。

（6）根据区块记载的交易内容，对其中的每项交易进行验证，确认交易均有效。

3.5　长链与短链

在全部的矿工创建区块的过程中，由于存在网络传输的延时，所以经常发生“分叉”现象，也就是出现两个矿工同时在系统中广播“自己完成了区块的哈希计算”并发布了新区块的现象，而且经过其他矿工的验证，他们的结果的确都是成立的。这时，系统中的矿工节点会各自选择一个区块作为父区块继续挖矿，但同时也保存另一个区块的信息作为“备用区块”，此时从系统的角度来看，意味着比特币区块链出现了分叉，切分为两条链，不同的矿工会随机选择不同的分支。

接下来，新的区块被计算出来，并链接到原来的分支链上。由于矿工选择是随机的，区块计算也有很大的不确定性，算力的分布也不是均匀的，所以在两条链上新增 1～2 个区块后必然会出现某个分支上的区块数量更多，长度也比其他分支要长的情况。根据规则（总是选择工作量最多的区块链），所有的矿工都会选择更长的那条区块链进行下一轮区块计算，包括那些之前在

仅供非商业用途或交流学习使用

较短的链上从事区块计算的矿工。于是长链更长，短链不再更新，原来短链上面的区块也会被丢弃，组装短链区块的矿工也得不到比特币的“奖励”。最重要的是，从系统的角度来看，比特币区块链又回到了只有唯一一条主链的情形，如图 3.9 所示。

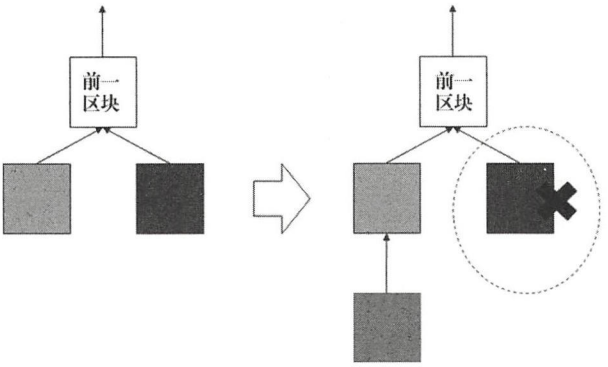


图 3.9　区块链分叉及修复示意图

TIPS:

根据比特币的历史经验，发生过的最多的平行区块链是 6 层，即两条链上各有 6 个区块继续同时产生。这种情况只发生过一次。

如果区块产生的时间间隔设计得更短，则可以更快完成交易的确认（比如，每分钟一个区块，就会比目前每 10 分钟一个区块的确认速度快 10 倍），但同时会导致区块链分叉更加频繁地出现。反之，把区块的交易组装时长设计为更长的间隔，由于区块内含交易的增多使区块的计算变得复杂，能够减少“同时形成两个有效区块”的分叉数量，降低了分叉的不利影响，但同时会导致更长的交易确认时间，影响用户的使用体验。所以，中本聪将比特币区块间隔设计为 10 分钟，是在更快速的交易确认和更低的分叉概率之间做出的平衡。

为了使区块产生的时间稳定在 10 分钟左右，比特币系统引入了哈希计算的难度值（Difficulty）自动调整机制，每当系统内完成了 2016 个区块的创建时，就会自动调整挖矿算法的难度系数。这个调整是由矿工节点按统一的公式自动计算的，因为每隔 10 分钟产生一个区块，产生 2016 个区块要花费 14 天（2 周），因此每隔 2 周左右比特币挖矿的难度会产生一次变动。

仅供非商业用途或交流学习使用

3.6　比特币的安全性

哈希（Hash）算法是一种将任意长度的消息压缩到某一固定长度（消息摘要）的函数，它的特点是：①从结果反推输入信息的内容，在计算理论上是基本不可能的；②两个不同的输入信息产生同样的哈希结果，这在计算理论上也是基本不可能的。因此，哈希算法具有不可逆性（不能从结果逆推出输入内容）、唯一性（一个输入内容只有一个哈希结果），可用于数字签名、消息的完整性检测、消息起源的认证检测等。由于哈希算法的不可逆性及唯一性，它被中本聪选择用作比特币的主要算法。在比特币区块的构造过程中，不断地对交易信息进行“两两哈希”，最终的默克根代表了所有交易的全部信息，即便这些交易中只是一个小小的字节被修改，也将导致其哈希值的改变，从而向上传导，直至最终默克根的改变。

TIPS:

假设有一个恶意的参与者，他想要篡改过去的某项交易，那么他需要怎么做呢？首先，他要进入该交易所在的区块，把原始交易改成自己想要的形式，然后计算这项交易的哈希值。接下来，他需要按照金字塔原则（两两哈希），逐级向上修改每层的哈希值，一直修改到最终的默克根。

这还没有完成，由于本区块默克根的改变，导致本区块的 ID 也会发生改变。因为本区块的 ID 是由“前一区块的 ID”“本区块的默克根”“随机数”三项进行 SHA256 哈希计算得到的，其中一项发生了变动，必然导致结果——本区块 ID 发生变动。所以，这个恶意的参与者还需要修改本区块的 ID。

当他篡改了本区块的 ID 时，又会引发下一区块的 ID 的改变。因为区块链的原理就是：下一区块在哈希计算本区块的 ID 时，其中的一项参数是“前一区块的 ID”，于是就产生了下一区块的 ID 变动。此时，如果篡改者的算力不足，那么他伪造的这个区块无法链接到原来的区块链上，因为这一区块的后面没有别的区块，它会被视作短链而被删除。

假设篡改者的算力非常强大，那么他需要修改下一区块的 ID，并进一步修改下一区块的下一区块的 ID……直到他修改的这条链的长度超过系统中主链的长度，他才能用自己篡改的区块链去替代系统原有的主链。而具备这样强大

仅供非商业用途或交流学习使用

算力的比特币“矿工”目前基本上没有（如果有，也是拥有超级计算机的大型机构等，它们不太可能应用超级计算机来篡改某个比特币区块的交易），而且从利益的角度来说，具备这样强大算力的矿工进行挖矿，可以获得比篡改交易区块链大得多的利益，因此他也没有利益驱动来实施交易伪造。这也说明，比特币的区块链底层技术有效地保证了交易无法被篡改。

在比特币挖矿中，执行哈希算法的能力可以作为矿工节点的算力标志，因为一方面要对所有的交易信息进行哈希，另一方面要用哈希算法计算出一个满足难度要求的结果以创建区块。随着比特币的发展，矿工的算力也在飞速扩张。如图 3.10 所示为比特币系统哈希速率曲线，粗略估算当前 37EHash/s 的哈希速率与 2013 年 7 月的平均 223THash/s 的哈希速率相比增长了 16.5 万倍，是比特币初期（2009 年）的哈希速率的 14 800 亿倍。

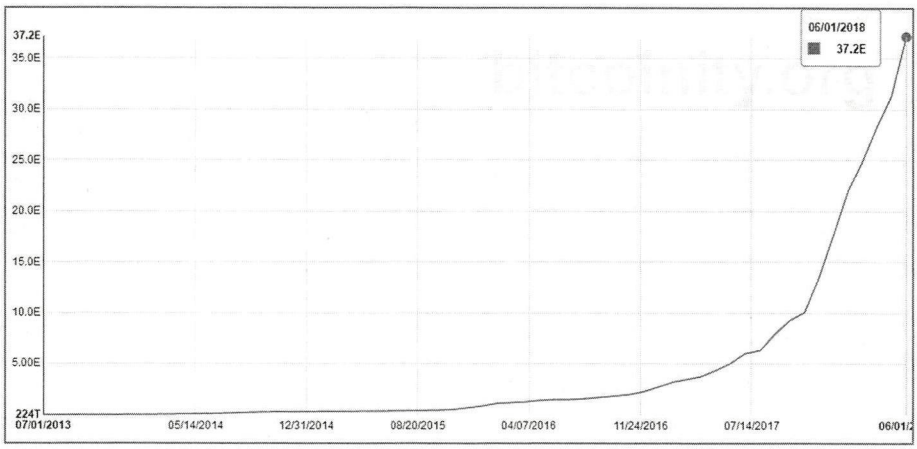


图 3.10　比特币系统哈希速率曲线

3.7　比特币挖矿设备的发展

从比特币的运行原理中可以看出，在比特币这样一个分布式的网络中，要想获得更多的比特币，就需要有更强大的算力，而且随着参与的人越多，对于算力的竞争要求就会越高，因为规则就是：最先完成区块哈希的矿工才能得到全网认可，取得新的比特币。

TIPS:

为了比竞争对手跑得更快、得到更多的比特币，矿工很快就进入哈希算力的“军备竞赛”之中，通过不断升级硬件来提升算力。

3.7.1　比特币挖矿设备的发展阶段

在比特币创建之初，中本聪仅使用个人计算机“挖”出了创世区块，最初的参与者都是使用个人计算机（利用 CPU 的算力）来进行挖矿的，也都能成功地挖到比特币。后来由于竞争加剧，仅利用 CPU 的算力是不够的，人们又加入了 GPU（图形处理器）的算力，因为 CPU 的优势在于逻辑能力，而 GPU 在浮点计算方面能力更强。接下来，根据算法的要求，有人专门开发出了矿机，从此挖矿基本上与普通人无缘了，就属于专业机构发挥规模优势的领域了。

随着比特币价格的上涨，吸引的参与人数越来越多，硬件提升的速度逐渐逼近“摩尔定律”的极限，矿工又变革了挖矿模式，从单人挖矿进入集群挖矿的时代。

回顾比特币挖矿历史，比特币挖矿共经历了以下 5 个时代[①]：

CPU 挖矿→GPU 挖矿→FPGA 挖矿→ASIC 挖矿→大规模集群挖矿

在挖矿芯片更新换代的同时，带来的挖矿速率的变化是：

CPU（20MHash/s）→GPU（400MHash/s）→FPGA（25GHash/s）→ASIC（3.5THash/s）→大规模集群挖矿（3.5THash/s 的 n 倍速，n 为矿机集群数量）

上面所说的挖矿速率的单位是计算机每秒产生 Hash 碰撞的能力，根据比特币的区块组装规则，需要不停地进行哈希计算，通过计算机随机哈希计算出结果去碰撞给定的难度目标，这个过程是没有投机取巧的方法的，只能一个一个哈希值地计算和对比，因此每秒运行哈希计算的速度越高，最终完成区块组装、获得比特币奖励的机会越大，这就导致矿工不断追寻提升哈希速度的硬件和软件解决方案，产生了专业挖矿的矿机，并且不断升级硬件芯片，逼近硬件芯片的速度极限；另外就是软件方面的改进，从单人挖矿进化到现在广泛使用

①参考：http://www.zhihu.com/question/20792042。

的矿池方案。在目前的比特币算力环境下，如果使用个人计算机、图形卡挖矿，在理论上是有机会挖到比特币的，但机会渺茫。

总的来说，挖矿芯片种类的发展经历了从 CPU（中央处理器）挖矿到 GPU（图形处理器，相对于 CPU，具有更强的运算能力）挖矿，再到 FPGA（现场可编程门阵列）挖矿，目前已经进入 ASIC（专用集成电路）挖矿时代。在数量上，挖矿方式也从最初的台式机、笔记本电脑单机挖矿，发展到用 1～2 台专门的矿机挖矿，以及用几十台矿机组合挖矿的小作坊，目前已经发展到利用大规模矿场（由数千台甚至上万台专业矿机组合）、矿池进行云挖矿的时代。

3.7.2 矿机与矿场

矿机是专门设计用来进行数字货币挖矿的硬件设备，与普通的个人计算机有较大区别，它的性能单一，只能进行哈希计算，但是专注于高速完成，所以计算性能优越，但一般还需另行外置软件控制系统。此外，矿机的运作需要消耗大量的电力，所以硬件散热、电力消耗也成为影响矿工收益的重要因素。如图 3.11 所示为“比特大陆”网站销售的单台 ASIC 矿机——蚂蚁矿机 S9i 14T，其主要技术参数为：额定算力为 14THash/s，功耗为 1320W。

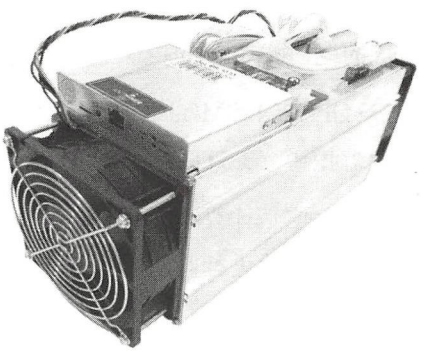


图 3.11 ASIC 矿机样品

一台矿机及其辅助系统的标准组成为：一台矿机、一个 AUC、一个树莓派（Raspberry Pi）、电源及各种连接线等，并通过计算机连入比特币网络。各种设备的连接顺序为：网线→树莓派→MicroUSB 线→AUC→4PIN 连接线→矿机和电源。

主要设备的功能如下。

（1）矿机：进行挖矿运算的主要硬件设备，采用专门为挖矿而设计的集成电路板（ASIC），大幅提高了专用运算能力。

（2）树莓派（控制器）：存储控制矿机运行的程序（因挖矿只需进行简单、高效的数值计算）并控制矿机的运行。每个树莓派可以控制约 60 台矿机。

（3）AUC：转换器，1 个 AUC 可以串联多台矿机。

单台矿机并联后可以组成小型挖矿作坊。由于使用的电力大幅提升，因此还需要配备变压器等电力辅助设备。如图 3.12 所示为 60 台矿机并联配置的连接示意图。

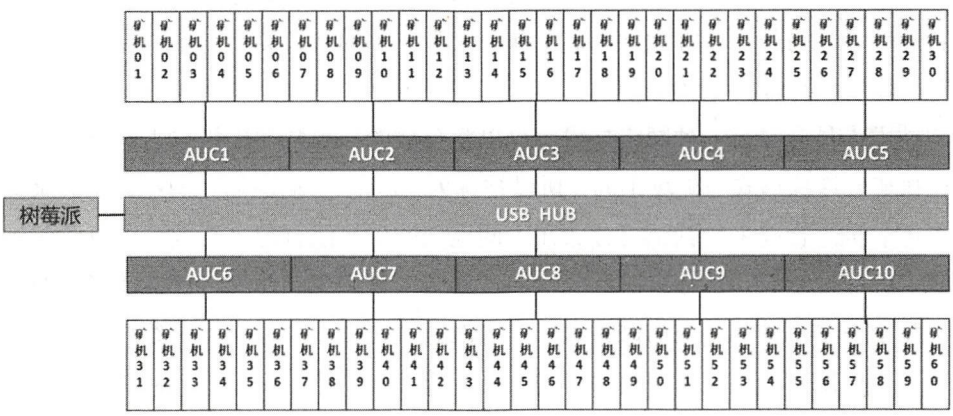


图 3.12　60 台矿机并联配置的连接示意图

随着比特币挖矿难度的大幅提升，以及对挖矿时间越来越快的需要，算力的提升成为保证挖矿成功率的最现实的追求，在短时间内矿工就进入大规模集群挖矿的时代，出现了大规模集群矿场。

TIPS:

由于矿机的电力消耗巨大，且大量矿机集中时的散热要求高，所以中国的矿场一般位于电力价格低、气温偏低的区域，比如中国西南部的云贵川地区（水电资源丰富、电力价格较低），或者内蒙古、东北等地（气温低、散热好）。一般在矿场内会布置数千台矿机。

影响挖矿收益的因素有很多，比如全网的算力和难度、矿机的性能和功耗、

矿场的部署和运维能力、廉价电力供应，以及币价和政策的导向等。而随着大型矿场越来越多，那些依靠单台矿机或几十台矿机组成的挖矿作坊的矿工，其盈利空间非常有限，挖矿逐渐成为大型企业或团队的游戏，因为它们可以取得廉价的电力供应、较强的芯片研发能力、专业的矿场部署和运营经验等。相对于单机或小规模的矿场，大型企业的这些优势必然会产生更强大的算力和更优化的计算速度，单位成本更低，最终将会最大化挖矿收益。

3.7.3 矿池

随着矿机的核心硬件——芯片的不断升级，最终它会发展到与当前计算机芯片的制作水平相当，再往下发展就会遇到天花板，没有办法再进一步提高哈希速率了。另外，单打独斗的矿工在与大型矿场的组合算力竞争中基本上也没有了获胜的可能，因此分散的矿工开始组团挖矿。来自比特币论坛 Bitcointalk 的技术人员开发了一种网络技术，可以将分散的、单独的矿机算力联合起来进行挖矿，这种形式与矿场不同，所以被称为“矿池”。矿池通过网络，将大量的单体矿机组合起来，甚至将分布在不同地域的矿机组合起来形成一个具备强大算力的矿机组织，进行挖矿计算，通过集合大量矿机的算力，使矿池能够在挖矿竞争中取得胜利、得到比特币。在矿池获得比特币后，将根据参与矿池的各个矿工在挖矿过程中所做出的算力贡献，按不同比例进行分配。随着比特币价格的不断上涨，参与矿池的矿工虽然每个人分到的比特币不多，但是总的收益还是可观的。

矿池的存在还由于单个比特币是可以再分割的。比特币的最小单位是 Satoshi（聪），1 个比特币=1 亿聪。因此，当矿池挖矿成功得到 12.5 个比特币后，它是足够在矿池内部按一定的规则进行比特币的分配的。如图 3.13 所示是矿池按算力比例分配奖励的示意图。为了便于计算，按每个区块链奖励 50 个比特币来说明（实际当前奖励情况是 12.5 个比特币/区块）。

矿池已经脱离了实体硬件的范围，扩展到虚拟的网络世界，它是由在不同区块的矿机通过软件、网络进行的算力的集中，我们已经无法看到矿池的物理存在。

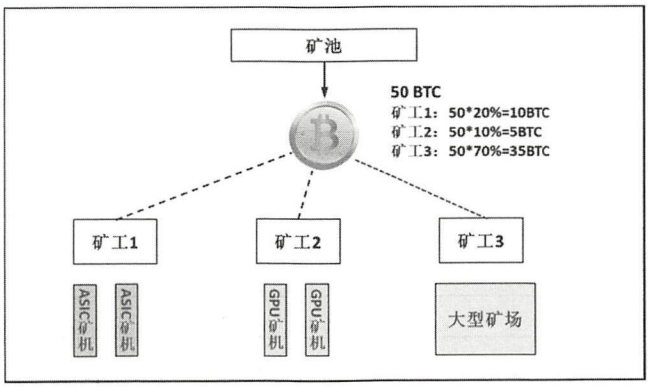


图 3.13　矿池按算力比例分配奖励的示意图

矿池的原理是：在当前水平下，比特币全网每 10 分钟产生一个区块，这个区块包含 12.5 个比特币。假设全球同时有 1 万人参与挖矿，但在这 10 分钟内，只有 1 个幸运儿依靠算力优势构造了区块，获得了这 12.5 个比特币，其他人则没有收益。如果单个矿工通过矿池形式组队开采，并按事先约定的分配方式分配所得到的比特币，由于矿池的算力优势，很容易首先构造区块成功并获得 12.5 个比特币，这 12.5 个比特币由矿池的参与者按比例分配，那么参与矿池的矿工的收益就会从单打独斗的零收益变成了按比例分配获得的比特币，收益大幅提升。目前矿池最常用的分配方式为 PPS 分配，就是按矿工自己提供的算力占矿池总体算力的比例来分配收益的。如果单个矿工的算力为 10T，整个矿池的算力是 100T，则单个矿工的算力占矿池总算力的 10%，如果矿池得到了 12.5 个比特币，那么该矿工可以获得 12.5×10%=1.25 个比特币。

TIPS:

目前，全世界有 22 家主要的矿池，已经占到了全网算力的 99%以上。为了平滑挖矿收益，大大小小的矿场都加入矿池之中。

根据 2017 年下半年的统计，这 22 家矿池的算力占比如表 3.1 所示。

表 3.1　全世界 22 家矿池的算力占比

矿池名称	所占比例
AntPool（蚂蚁矿池）	25.2%
BTC.TOP	11.2%
BTC.COM	10.2%

续表

矿池名称	所占比例
Bixin	8.1%
BTCC Pool	7.3%
F2Pool	5.4%
ViaBTC	5.0%
BW.COM	4.9%
BitClub Network	4.8%
Bitcoin.com	3%
SlushPool	2.6%
GBMiners	2%
BitFury	1.9%
CANOE	1.2%
1HASH	1.1%
Telco 214	1.1%
BATPOOL	0.9%
Waterhole	0.6%
ConnectBTC	0.6%
Bitcoin India	0.2%
Solo CKPool	0.2%
BitcoinRussia	0.2%

在表 3.1 中，排名前 8 位的矿池，如 AntPool（蚂蚁矿池）、BTC.TOP、BTC.COM、Bixin、BTCC Poll、F2Pool 等，都是位于中国的矿池，占据了全网算力的 80%左右。当然，为了保证不同国家和地区的矿工能够顺利接入，这些矿池在矿工数量较多的不同国家和地区都架设有服务器，以确保矿工接入速度、保证挖矿收益。

3.7.4 云挖矿的应用

随着挖矿过程中算力竞争的加剧，挖矿需要消耗越来越多的电费，成本相当高。同时，运行大规模的矿场、系统的安装与维护等要求矿工投入大量的金钱与精力，也使得中小矿工甚至个人基本上没有可能参与到挖矿中并取得比特币收益。矿机的价格越来越高，普通人想参与挖矿也要付出一笔不菲的矿机采购费用；并且矿机的系统维护也需要专业水平，而普通人并没有这方面的能力或资源。

云挖矿服务的出现，使普通人也可以参与到比特币挖矿中来。

TIPS:

所谓云挖矿，是指一些投资者在电费低廉的地区（如中国的云贵川、内蒙古等地）大规模建设矿场。由于这些矿场拥有更低的机器价格和低廉的电费，还可以通过机房结构的设计和当地的地理及气温优势来实现良好的散热通风，降低机器消耗的能量，所以具备很强的算力。

这些投资者再在网上向各地的普通用户出租算力的硬件资源，用户通过付费购买或租赁这些大规模矿场的部分算力，取得了按比例参与分配矿场取得的比特币的权益。矿场的投资者也通过收取的购买或租赁算力费用，实现了投资回报。云挖矿的应用产生了比特币挖矿的新盈利模式。

3.8　比特币交易中的非对称加密

在比特币交易中，为了保证交易的安全性，采用了一种比较复杂的加密机制：公钥与私钥[①]。公钥（Public Key）与私钥（Private Key）加密属于“非对称加密”方式，它们总是成对出现、互相解密的，当其中一个用于加密时，只能用另一个来解密，而且两个密钥在数学上相关，用私钥可以计算出公钥，但反过来用公钥不能计算出私钥，所以称为“非对称加密”。与之相对的是“对称加密”方式，是指交易双方都知道同一个密码，用该密码来对交易进行解密，比如常用的对压缩文件的加密和解密。这两种加密方式对比来看，非对称加密的安全性更高，但是效率会受到影响；对称加密的效率更高，可以用于大量文件的快速解密，但是一旦密码泄露，则容易导致损失。

3.8.1　非对称加密原理

关于公钥与私钥配对的非对称加密原理，我们可以打一个比方来说明。比

① 参考：http://8btc.com/article-1756-1.html。

如，有两个用户 A 和 B，每个人都设置了自己账户的公钥和私钥，其中，公钥是公开向外发布的、让对方接收的；私钥是只有自己才掌握的，用于接收信息时的解密，以及数字签名。当 A 发送信息给 B 时，他先用 B 发过来的公钥对内容进行加密，再发送给 B，即使在传输过程中有人截获了 A 所发的信息，由于没有 B 的私钥也无法解密内容；而 B 接收到 A 发来的信息后，使用自己的私钥进行解密，就可以知道 A 所发信息的内容。

同样，B 要给 A 发送信息，先用 A 的公钥对内容进行加密，A 接收到信息后用只有自己才掌握的私钥进行解密，这样双方就实现了保密的信息传送。整个加密和解密过程如图 3.14 所示。

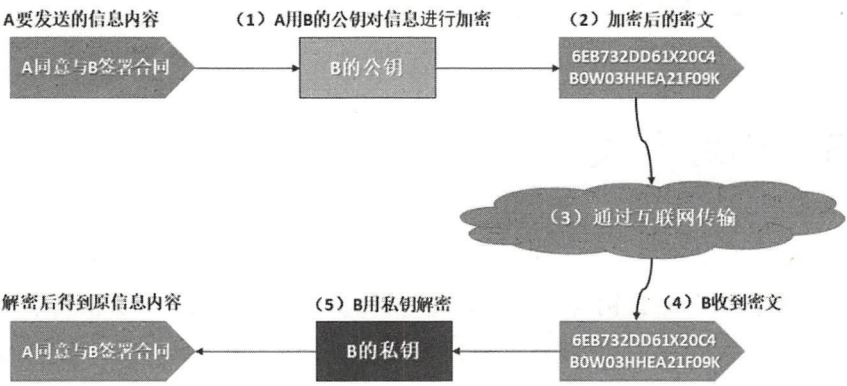


图 3.14　公钥与私钥的加密和解密过程

另外，私钥还可以用来进行数字签名。比如，A 发送给 B 一段信息，使用了自己的私钥加密，对于 B 来说，收到后只有使用 A 的公钥才能解密，用其他人的公钥则无法解密，这就相当于确认了信息是由 A 发来的，而不是由 C、D 或其他人发来的，对信息的来源构成了身份验证的数字签名。

在比特币交易中，也采用了这种公钥与私钥加密机制。私钥在形式上是一段字符串，如 5KYZdUEo39z3FPrtuX2QbbwGnNP5zTd7yyr2SC1j299sBCnWjss 就是一个私钥。所有支持比特币协议的应用在看到这段字符串后都可以将其识别为私钥，并把私钥转换成公钥，再转换为账户地址。如果对应的账户上有比特币，就可以使用这个私钥来使用账户中存储的比特币。

TIPS:

公钥和地址的生成都依赖私钥，所以私钥在比特币交易中非常重要，需要采用各种方法保管好，防止泄露。私钥一旦泄露或被黑客窃取，而比特币系统又只认密码不认人，就会导致比特币的损失。

私钥是由随机函数生成的 32 字节组成的数组，1 字节等于 8 位二进制数字，一个二进制数字为 0 或者 1，所以全部私钥组合的总数是 $2^{(8\times32)}=2^{256}$ 个，这个数量已经超过了宇宙中原子的总数，想要通过遍历所有的私钥组合来试图找出加密用的私钥，这种暴力破解方法在目前算力下成功的概率是非常低的，所以私钥从密码学上来看是非常安全的。但是私钥是由随机函数产生的，它对于随机函数的要求比较高，要求随机产生的结果是不可预测、不可遍历的，这样才能使私钥不会因为太容易耦合而造成比特币持有者的损失。

真正的私钥是由 32 字节组成的数组，由 256 个 0 或者 1 组成，原样显示太长，而且不便于识别。为了便于显示和识别，一般在使用时会对私钥的 32 字节的数组进行 Base58 转换，将 256 位的私钥转换为 58 位的形式。Base58 是比特币中使用的一种独特的编码方式，主要用于产生比特币的钱包地址和公钥。

公钥是由私钥通过椭圆曲线（Elliptic Curves）生成的，一个私钥经过椭圆曲线变换之后会生成一个 65 字节的数组，比如这样一个数组：04a34b99f22c790c4e36b2b3c2c35a36db06226e41c692fc82b8b56ac1c540c5bd5b8dec5235a0fa8722476c7709c02559e3aa73aa03918ba2d492eea75abea235，它就是由私钥生成的公钥。为了便于显示，公钥一般会把字节数组经过 HEX（十六进制）处理之后显示出来，这与私钥所用的 Base58 编码显示方式不同。公钥的作用是解开私钥签名的数据，交易者在使用私钥对交易进行签名之后，会把交易与自己的公钥一起发送；经过私钥加密（数字签名）的数据可以使用公钥解密，矿工接收到公钥之后，通过解密交易，验证私钥的签名的正确性，确认交易方的身份。由于私钥和公钥是对应关系，用一个私钥签名的数据，只有对应的公钥才能解开，账户地址也是从私钥生成的，因此接收到交易广播信息的矿工通过公钥解密交易，并验证公钥对应的账户地址，就可以完成对交易的验证和对支付者身份的确认。同时，矿工并未掌握用户的私钥，也无法从公钥反推出私钥，所以用户的账户及资产不会受到影响。

3.8.2 生成钱包地址

地址是由公钥产生的，在比特币网络中看到的地址都是经过 Base58 编码处理的。地址的生成过程是：先对公钥进行一次 SHA256 计算，再对得到的结果进行一次 RIPEMD160 计算，最后从结果中取出 20 字节的数组，这个数组就是得到的 hash160 值，如 9a1c78a507689f6f54b847ad1cef1e614ee23f1e；在这个 20 位的字节数组前面再加上 1 字节（地址的版本信息），得到一个 21 位的字节数组，然后对这个数组进行 Base58 转换，就生成了钱包的地址信息，如图 3.15 所示。

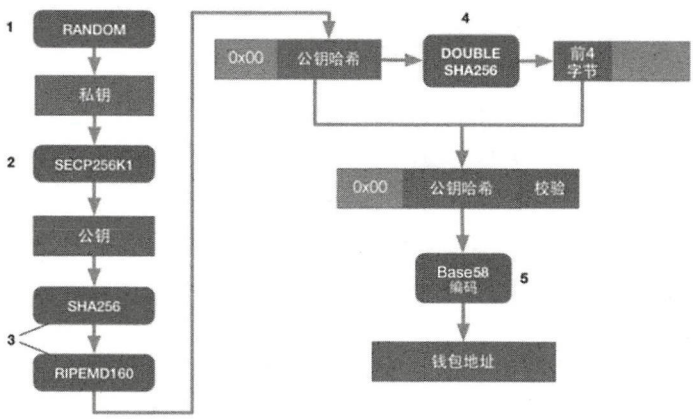


图 3.15 钱包地址的生成过程

过程描述如下：

（1）使用随机数发生器生成一个“私钥”。一般来说，这是一个 256bits 的数字，拥有了这串数字就可以对相应“钱包地址”中的比特币进行操作，所以“私钥”必须被安全地保存起来。

（2）“私钥”经过 SECP256K1 算法处理生成了“公钥”。SECP256K1 是一种椭圆曲线算法，通过一个已知“私钥”可以算得“公钥”，而在“公钥”已知时却无法反向计算出“私钥”。这是保障比特币安全的算法基础。

（3）同 SHA256 一样，RIPEMD160 也是一种哈希算法，由“公钥”可以计算得到“公钥哈希”，而反过来是行不通的。

（4）将 1 字节的地址版本号连接到“公钥哈希”头部（对于比特币网络的

公钥地址来说，这 1 字节为“0”），然后对其进行两次 SHA256 运算，将结果的前 4 字节作为“公钥哈希”的校验值，连接在其尾部。

（5）将上一步得到的结果使用 Base58 进行编码（比特币定制版本），就得到了“钱包地址”，比如 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa。

在上述的 5 个步骤里只有“Base58 编码”有相应的可逆算法（Base58 解码），其他算法都是不可逆的，所以这些数据之间的关系可以表示为图 3.16。

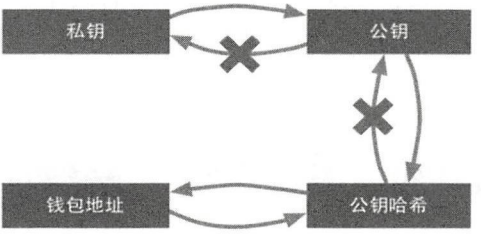


图 3.16　私钥、公钥、公钥哈希与钱包地址的关系

3.8.3　交易加密过程

比特币钱包间的转账是通过交易（Transaction）来实现的。交易数据是由转出钱包“私钥”的所有者生成的，也就是说，有了“私钥”就可以花费该钱包里的比特币余额。私钥加密交易过程示意图如图 3.17 所示。

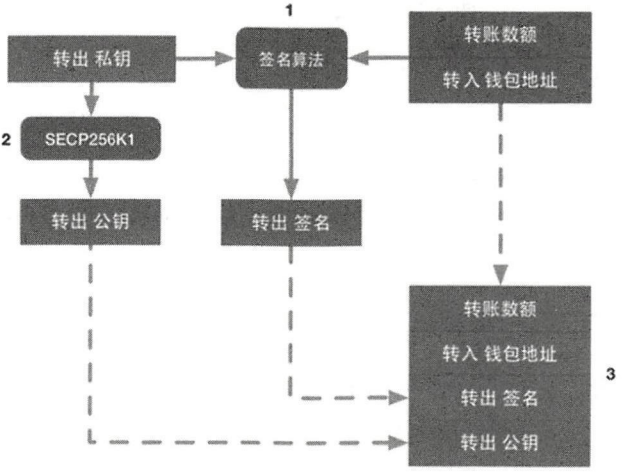


图 3.17　私钥加密交易过程示意图

过程描述如下：

（1）交易的原始数据包括“转账数额”和“转入钱包地址”，同时为了证明交易的生成者对“转出钱包地址”余额有动用的权利，需要用交易发起者的“私钥”对原始数据进行签名（加密）。

（2）生成“转出公钥”，即对私钥进行椭圆曲线算法处理，生成公钥地址。

（3）将“转出签名”和“转出公钥”添加到原始交易数据中，生成正式的交易数据，这样它就可以被广播到比特币网络进行转账了。

接下来使用“公钥”对签名进行验证，如图 3.18 所示。

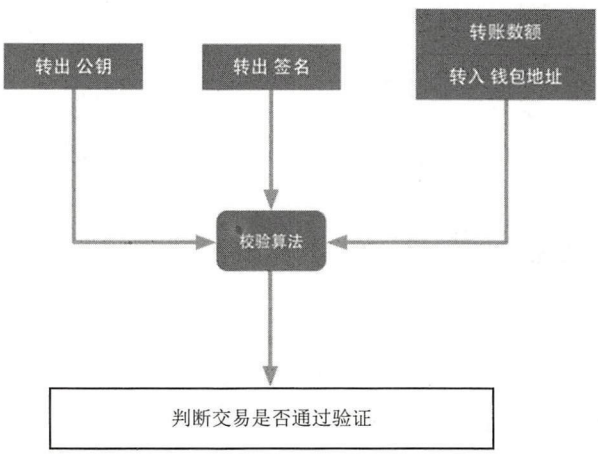


图 3.18　用公钥校验交易示意图

交易数据被广播到比特币网络后，网络中的节点会对这个交易数据进行检验，其中就包括对签名的校验，即采用发出的公钥对交易进行解密。如果校验正确，交易就可以被矿工确认，数据被写入区块链，得到全网络节点的共同认证，比特币余额就成功地从“转出钱包地址”转移到“转入钱包地址”，交易完成。

TIPS:

如果一个“钱包地址”发送余额到其他“钱包地址”，并被记入区块链中，那么它的“公钥”会被网络的全部节点所了解。但是由于公钥生成算法（SECP256K1）是不可逆的，即使“公钥”暴露，也不能逆向推导出私钥，因此私钥具有很高的安全性。

“私钥”用来生成“公钥”和“钱包地址”，同时也可以对交易进行数字加密，加密后发送到交易对方，由对方用“公钥”解密，交易得以完成。从私钥到公钥再到钱包地址的衍生过程说明私钥对于持有比特币的钱包来说是最重要的，因为拥有私钥就相当于拥有了钱包中的资产，所以，每款数字货币等与比特币类似的区块链应用，都应当首先实现对“私钥”安全的充分保护。

3.9　从比特币到区块链

比特币起源于 2009 年，从某种意义上来说，比特币网络类似于一种无组织的跨国支付体系。这个网络是不属于任何政府或者金融机构的，只是一个由散布在全球的计算机创建出来的电脑网络，它依靠全网络节点的共同参与和对总账本的维护来保持系统的健康运行。

通过在全球范围内对比特币进行广泛研究，人们逐渐认识到它所带来的底层技术——区块链技术的重大意义，比特币的应用也使区块链的各个关键组成部分，如分布式账户、全网维护、工作量证明的“挖矿”机制（算法各有不同），以及密码学技术的大量使用（公钥、私钥和哈希算法等），得到了全球范围内的验证，证明了区块链这种技术存在的颠覆性潜力。

区块链提供了一种新的思路，也带来了广阔的空间，从理念上来看它具备颠覆性的力量，但是目前我们只能预感到这种力量。要实现这种力量，区块链的建设和发展还需要取得更多的进展，首先需要做到的是对区块链的底层进行研究和开发。

目前已经被广泛接受的认识是：区块链的底层是一种全球性的、开放和透明的分布式节点（P2P）网络，这个网络需要建立起来并互联互通。打一个比方，这个网络是遍布全球的高速公路网，用户只要有车，就可以在高速公路网上开到全球任何地方。这个网络是全球性区块链的基础设施，既需要充足的硬件投入，也需要在运行机制上（共同的协议上）形成全球性的共识。一旦这一特点的区块链建立起来，它就具备了在区块链层面上达到“世界语”的高度。



TIPS:

从目前来看，初步具备全球性的底层网络的雏形的只有比特币和随后产生的以太坊，其中以太坊是在比特币之后创立，并主要针对比特币的不足之处开发出来的。

对于区块链而言，正如我们说到“互联网”这个名词一样，它也必然包含 3 个层次的结构。

（1）底层的全球开放式区块链。也许会有两个或多个全球范围内的开放式区块链，它们将是各自范围内全球所有交易的主链，其他的次级区块链最终都要链接到这些主链上；而这些底层区块链之间处于并行状态，它们通过共同认可的通用协议进行通信，彼此保持互信。

（2）次级区块链。比如某个行业自己构建的区块链，在行业内部形成分布式节点、共识机制、加密算法、激励机制等区块链要素，但是这样的区块链受限于一定的范围，不是其他的任意一个节点都可以进入的。同时，这类区块链通过通用协议与底层的全球开放式区块主链进行通信，将自己的交易信息写入区块的交易区，并通过类似默克根的形式进行不可破解式数据处理，在获得主区块链的共同认可后进入“总账”，最终成为主区块链的一部分。

（3）私人区块链。技术的发展将使个人与周围的物体（比如，智能电器、汽车等）之间、物与物之间构建成物联网，如果这种物联网是点对点（P2P）式的，那么它就可以应用区块链技术构建一条私有区块链，在各个节点之间形成去中心化的小型“总账”，促进交易的形成；与此同时，私人区块链也通过某一通用的协议链接到次级区块链或底层的主区块链上，将自身内部的交易信息写入区块并链接到主链上。

从前面介绍的比特币的工作机制中得知，区块链中区块的建立与交易过程其实是分离的，区块链网络中有一些参与者是专门从事区块的建立的，也就是比特币网络中的“矿工”，他们其实是区块链网络的公共服务机构，因为他们并不关心交易的细节，只是对获得节点共识机制确认后的交易进行封装、构建区块，他们需要获得激励，否则底层的主区块链就无人再来建立区块。因此，任何一条底层的、公开的区块链必须解决对于“公共服务机构”的奖励，为区块链的不断延长提供支持。



而对于次级区块链、私有区块链而言，由于存在利益相关，所以在其中既可以建立自己的“区块”和“区块链”，也能够设置内部的创建区块的服务机构。同时，由于系统范围的缩小，以及节点之间的互信机制远远高于全球公用区块链，因此在交易中的响应速度可以远远快于全球公用区块链，并且在共识机制的设计上可以采取更加高效、简捷的方式。

通过对比特币等区块链典型应用的介绍，我们可以得出区块链运行的流程图，如图 3.19 所示。

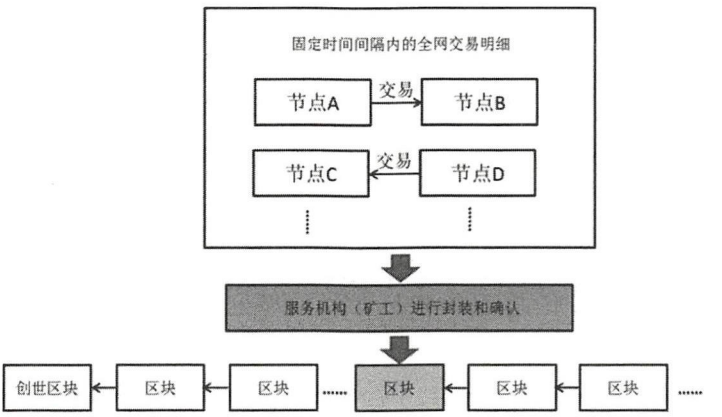


图 3.19　区块链运行流程图

由于区块链通过全网络节点的参与和维护解决了互联网的信任问题，以及具备无须权威中心机构认证的去中心化特性（从而大幅降低了交易成本）、历史数据不可篡改特性，并通过加密算法解决了安全性问题，所以它将成为互联网对“中心化”的传统行业进行颠覆的最强有力的方式，在大量需要信任才能交易的领域，如银行信贷和融资、支付及结算、保险、证券交易、供应链、合约签署、权属登记和转让、投票等领域都大有可为。

3.10　区块链的系统框架

从本质上来看，区块链是一个互联网产品，它是在互联网基础上建立起来的“去中心化、分布式”网络账本，它的架构也具备网络的相关特点，其中最



基础、底层的是数据结构，即信息和数据采取什么样的格式组织起来，输入区块链系统并最终由区块链系统进行处理。只有各个网络节点都采用统一的数据格式，信息才能无障碍地在这个网络中传输和交流。

当统一格式的数据进入区块链系统后，通过网络层进行连接和传输，在全网的节点之间进行广播、验证，这是区块链“去中心化”的基础条件和必备流程。

区块链的另一个关键要素是共识机制的达成，因为交易的确认、区块的构建这些基本功能都是在由全网节点采取一种共识机制达成一致后才能实现的。

TIPS:

这一层面也是最消耗算力资源的过程。以比特币为例，无数的矿机设备、电力投入其中，就是为了完成算力竞争并得到全网的共同认可，才能创建区块。

所有的区块进行链接“组装”，最终构筑出区块链的各种平台，包括公有链、联盟链及私有链。而诸如比特币、智能合约等，都是在区块链平台上运行的应用产品，是面对用户的最终产品，用户无须关注底层的构造逻辑，只需熟悉应用产品即可。这就是区块链的系统框架，如图 3.20 所示。

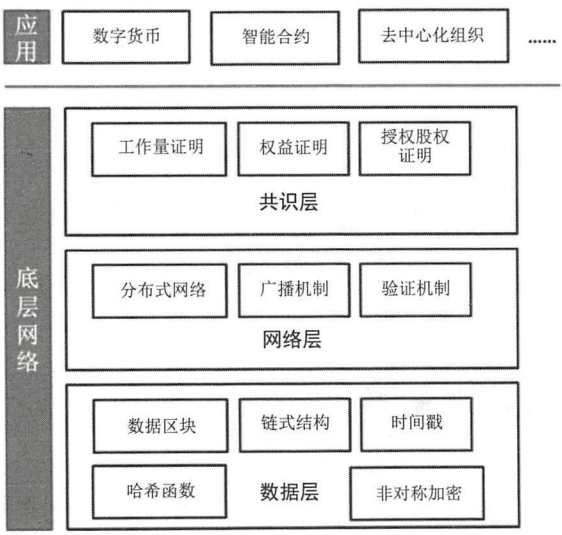


图 3.20　区块链的系统框架



在区块链的系统框架中，各层之间并不是割裂的，而是构造了一个以交易为中心的完整循环结构。首先，应用产品在面向底层网络时传输的是数据，所以应用层相当于数据层的输入源。无论是数字货币还是其他产品，如果要引入区块链中，那么其数据必须进入数据层，按照区块链的格式进行封装。

其次，当数据封装完成后，就进入 P2P 网络进行全网广播，由全网的服务机构（比如比特币的“矿工”）进行确认，采用工作量证明机制。

最后，当获得全网的确认，即形成了“共识”后，区块构造完成，并链接到主链上，完成了一次交易的流程，由应用层开始启动下一次交易的循环。

3.11　分布式网络

区块链技术是在网络技术快速发展之后出现的产物，它不是无源之水、无根之本，而是在分布式网络的基础上发展出来的，也可以说是分布式网络的一种新型应用。了解分布式网络技术的特点，也有助于我们进一步了解区块链技术。

3.11.1　分布式网络的概念

区块链建立的物理网络基础是点对点的分布式网络，与传统的中心化“客户端/服务器”网络架构有很大的不同，它是一种去中心化的网络结构方式，提高了数据传输的效率，相比中心化的网络结构而言具有更高的安全性。

作为区块链的底层物理网络基础，这种分布式对等网络又称为 Peer to Peer（简称 P2P 网络），其最主要的特点是：每个网络节点之间都是平等的，没有哪个节点处于中心地位或者对其他节点具有控制、管理权限。

在 P2P 网络环境中，彼此连接的多台计算机之间都处于平等的地位，各台计算机都有相同的功能，无主从之分，网络中的每台计算机既能充当网络服务的请求者，又对其他计算机的请求做出响应，提供资源、服务和内容，无须由中央服务器来控制和提供。通常这些资源和服务包括信息的共享和交换、计算



仅供非商业用途或交流学习使用

资源（如 CPU 计算能力）共享、存储（如缓存和磁盘空间的使用）共享、网络共享、打印机共享等。

下面我们用一张网络结构图（见图 3.21）来表示分布式网络与传统的中心化网络的区别：有无一个网络节点的中心。

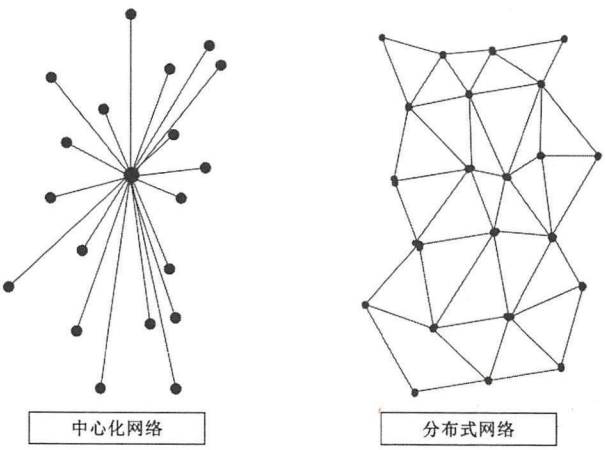


图 3.21　中心化网络结构与分布式网络结构

3.11.2　分布式网络的特点

分布式网络的特点体现在以下几个方面。

（1）非中心化：网络中的资源和服务分散在所有节点上，信息的传输和服务的实现都直接在节点之间进行，无须中间环节和服务器的介入，避免了可能的瓶颈。所有的节点之间在逻辑上是平等的。P2P 的非中心化基本特点带来了其在可扩展性、健壮性等方面的优势。

（2）可扩展性：在 P2P 网络中，随着用户的加入，不仅服务的需求增加了，系统整体的资源和服务能力也在同步扩充，始终能比较容易地满足用户的需要。理论上其可扩展性几乎可以被认为是无限的。例如，在传统的通过 FTP 的文件下载方式中，当下载用户增加之后，下载速度会变得越来越慢；然而 P2P 网络正好相反，加入的用户越多，P2P 网络中提供资源的节点就越多，其他用户的选择和网络通道相应地也越多，下载速度反而越快。

（3）健壮性：P2P 网络比中心化网络更加健壮，具有耐攻击、高容错的优



仅供非商业用途或交流学习使用

点。由于服务是分散在各个节点之间进行的，信息也是分散存放在不同的节点上的，所以当部分节点或网络遭到破坏时，对其他部分的影响很小，也不会影响到整体网络的信息安全。P2P 网络一般在部分节点失效时能够自动调整整体信息传输通道，通过绕开失效节点，可以保持其他节点的连通性。P2P 网络通常是以自组织的方式建立起来的，并允许节点自由地加入和离开。

（4）高性价比：性能优势是 P2P 网络被广泛关注的一个重要原因。随着硬件技术的发展，个人计算机的计算和存储能力及网络带宽等性能依照摩尔定律高速增长。采用 P2P 网络架构可以有效地利用互联网中散布的大量普通节点，将计算任务或存储资料分布到所有节点上，利用其中闲置的计算能力或存储空间，达到高性能计算和海量存储的目的。同时，无须由中心节点进行大规模投资以提高网络的存储能力，从而降低了投入成本。

（5）隐私保护：在 P2P 网络中，由于信息的传输分散在各节点之间进行而无须经过某个集中环节，用户的隐私信息被窃听和泄漏的可能性大大缩小。此外，目前解决 Internet 隐私问题主要采用中继转发的技术方法，从而将通信的参与者隐藏在众多的网络实体之中。在传统的一些匿名通信系统中，实现这一机制依赖某些中继服务器节点。而在 P2P 网络中，所有参与者都可以提供中继转发的功能，因而大大提高了匿名通信的灵活性和可靠性，能够为用户提供更好的隐私保护。

（6）负载均衡：在 P2P 网络环境下，由于每个节点既是服务器又是客户端，因而降低了在传统 C/S 结构下对中心服务器计算能力、存储能力的要求。同时因为资源分布在多个节点上，因而更好地实现了整个网络的负载均衡。

正是由于 P2P 网络具备这样的特点，极大地满足了区块链技术的要求，因此区块链技术的物理网络基础必然以 P2P 网络为唯一架构。

3.11.3　分布式网络的两种架构

根据节点连接方式及资源定位方式的不同，分布式网络分为两种架构[①]：一种是非结构化的 P2P 网络；另一种是结构化的 P2P 网络。

① 参考：https://en.wikipedia.org/wiki/Peer-to-peer。



仅供非商业用途或交流学习使用

TIPS:

所谓非结构化的 P2P 网络，是指网络系统的各个节点随机地互相连接在一起，没有经过特定设计的层叠网络结构。其特点就是：搭建网络容易；另外，由于各节点在网络中的角色是相同的，所以面对网络扰动（如大量节点不断地加入或退出），网络本身可以保持相当的稳定性，不会受到影响。

非结构化的 P2P 网络有一个很大的缺陷，那就是数据索引的效率低。原因在于：当一个节点想要在网络中寻找到自己需要的数据时，由于整个网络缺乏结构，它不知道在哪个节点上存储着自己想要的数据，所以需要向尽可能多的节点发送寻找这项数据的请求；而且网络中的其他节点都要回应和处理这一请求，即存储有相关数据的节点要向请求者回复“此处有你需要的数据”，没有相关数据的节点要告诉请求者“我这里没有你想要的数据”。假设请求者节点向网络中的其他 100 个节点发送了数据请求，它也会收到 100 条回复，系统中就出现了 200 条信令（请求 100 条+回复 100 条）；如果同时有 n 个节点向网络请求查询数据，则形成的信令数量就是 $200n$ 条，很容易导致“信令洪水”的发生，消耗大量的 CPU 和带宽资源，用于大量节点之间的数据查询和通信。

TIPS:

如果请求者寻找的数据是比较流行的，比如热播电影或流行音乐，那么网络中大量的节点会存储这类数据，这时请求者的数据请求会很快得到回应；但如果请求者寻找的是比较小众的数据（本来只存储在网络中的少量节点上），那么请求者的数据请求失败的可能性就大大增加。

如图 3.22 所示是非结构化的 P2P 网络结构示意图。

结构化的 P2P 网络的特点是网络节点通过特定的网络拓扑结构连接在一起，通过网络协议确保任意节点可以高效寻找到所需的数据，即使很少见的数据。目前最常用的结构是 Distributed Hash Table（DHT），即分布式哈希表。如图 3.23 所示为采用 DHT 结构的 P2P 网络结构示意图。



仅供非商业用途或交流学习使用

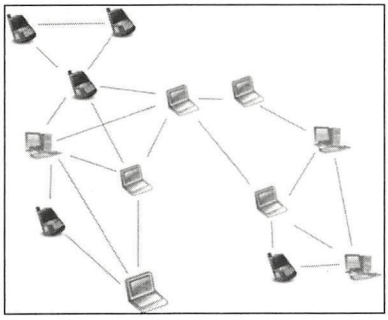


图 3.22　非结构化的 P2P 网络结构示意图

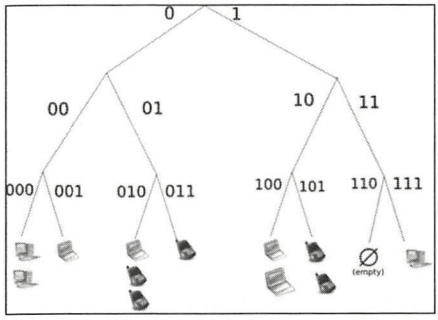


图 3.23　采用 DHT 结构的 P2P 网络结构示意图

TIPS:

在采用 DHT 结构的 P2P 网络中，采用了哈希表的方式来对应资源与存储空间，即：对于某一资源 KEY，首先使用哈希函数计算出这个 KEY 值的索引 index，即 hash(KEY)=index，然后把与这个 KEY 值对应的数据资源 VALUE 存储到 index 所对应的存储空间中，这样就建立了(KEY,VALUE)的对应关系。当每次想要查找 KEY 所对应的 VALUE 值时，只需进行哈希运算找到其 index 值并访问对应的存储空间，就可以找到所需数据。

为了提高搜索效率，网络中的每个节点都保存着一份网络节点的哈希表，通过这一哈希表使网络资源与存储节点对应起来。当需要查找某一资源时，先对该资源进行哈希计算，找到唯一对应的 index（索引值），然后访问这一索引值对应的节点或存储空间，就能很快找到所需的数据资源，减少了网络中“信令洪水”的发生，搜索的成功率也大大提升。

结构化 P2P 网络也有不足之处，那就是由于网络中的哈希表建立了资源与存储空间的对应关系，所以网络抗扰动的能力不足。比如，当大量节点经常加入或退出网络时，网络数据与资源的对应关系发生了很大变化，原来的哈希表的作用就会大幅降低，效率也随之下降。

3.12　广播与验证机制

广播与验证机制是区块链网络产生信任、形成高安全性的基础，也是区块链网络达成共识、形成全网共同维护账本的基础性工作。

仅供非商业用途或交流学习使用

3.12.1 广播

在区块链的分布式网络中存在两种广播机制：一种是交易广播机制，即对于每笔交易来说，需要向全网进行广播，取得全网中服务节点（如矿工）的验证，被矿工“打包”进入区块的构造过程；另一种是区块构造广播机制，即由网络服务机构完成的广播。当矿工进行了大量的计算，完成了工作量证明时，需要把自己的运算结果也广播到全部网络节点中，由其他节点根据给定的计算条件调动资源（算力）进行验证，当验证通过后，该区块成为全网节点认可的最新区块，所有矿工再以该区块为起点进入新的区块生产过程。

交易广播机制的存在使得区块链中的每项交易置于全网节点的监督之下，交易的每个细节都要受到其他节点的检验，如支付资金的来源是否可以追溯、支付过程是否符合规则、支付的结果是否确定等，一旦经过验证是一项真实的交易，那么这笔交易就进入了全网节点的区块内，由全网节点为它的真实性、合规性进行背书，无法篡改，同时下一项与之相关的衍生交易在它的基础上产生，也就无须另外中心机构的背书即可信任该项交易，这就是区块链网络信任的基础。

TIPS:

可以说，交易广播一方面是下一项交易的源头，是产生信任机制的第一步；另一方面是区块封装的数据基础，因为矿工要做的就是将某一时段内的交易组装为区块并进行计算。

区块构造广播，即矿工竞争构造区块的结果广播，是区块链运行的基础流程之一。区块的构造是由全网的服务机构（矿工）竞争的结果，在接收交易广播的信息，完成本时段内全部交易的封装后，矿工开始竞争计算 Hash 值，最快完成计算的矿工必须尽快将结果广播出去，让全网络内的其他矿工加以验证，才能使区块真正构建完成，实现区块向主区块链的“入链”过程。

3.12.2 验证

验证机制是指节点对广播的交易信息和区块进行验证的过程。区块链中没

仅供非商业用途或交流学习使用

有中心机构进行交易信息检验和保证，因此每项交易需要依靠全网节点的验证来保证，只有经过至少 51%节点验证的交易才能取得信任，成为区块封装中的交易组合信息。

对于网络节点来说，在接收到交易信息的广播后，最重要的验证是支付资金的来源是否可追溯。以比特币网络为例，张三向李四支付了 50 个比特币，当这笔交易广播出去后，其他节点（比如矿工）要检验这 50 个比特币的来源，是张三挖矿所得，还是他购买所得？这 50 个比特币是否已经花费过（避免重复使用）？通过查询 50 个比特币各自对应的前一交易地址，层层回溯，这两个重要的问题很容易找到答案。

假设其中 25 个比特币是张三挖矿所得，另外 25 个比特币是张三向其他节点购买所得，那么验证时还需要进一步回溯到挖矿的币基交易值、购买的哈希值是否来源于其他区块。需要说明的是，从理论上来讲，只要有时间和资源进行回溯，那么所有的交易都可以一直回溯到创世区块。

TIPS:

> 由于区块数量较大，并考虑交易的时效性、验证的经济性，所以节点对交易的验证过程并不关注过多的回溯和源交易的细节，一般只对源交易的哈希值进行验证，只需确认源交易的哈希值是可信的，而且存在于此前的区块中即可。

这是一种可行的提高效率的措施，因为在区块链的验证机制之下，源交易已经通过验证并形成了哈希值，无须再去层层溯源到创世区块，前面的所有来源都已经在历史交易过程中被验证过了。交易信息的其他验证，包括数据格式的验证、交易双方的数字签名等，都作为支付验证的辅助信息。只有通过了全网验证的交易，在得到了网络的一致认可后，被支付方才能真正“取得”支付的资金，用于自己的其他交易。

区块构造的验证也是非常重要的，以比特币网络为例，当最先完成区块封装及竞争计算的矿工把构造结果广播出去以后，其他节点的矿工就会停止对本区块的构造，转为验证该区块的计算结果是否成立。每个节点在将新的区块转发到其他节点之前，都会进行一系列的测试去验证它，从而确保了区块的合格

仅供非商业用途或交流学习使用

与有效。具体的操作是：矿工节点依据一份标准清单对该区块进行验证，这些标准存在于比特币核心客户端程序的 CheckBlock 和 CheckBlockHead 函数中，包括以下几个方面：①区块的数据结构在语法上有效；②区块头的哈希值小于目标难度（确认包含足够的工作量证明）；③区块时间戳早于验证时刻未来 2 小时（允许时间错误）；④区块大小在长度限制之内；⑤其他条件。

TIPS:

区块构造验证也是一种保证系统正常运行的激励机制，没有这种机制，区块链系统难以稳定运行。

在这种机制下，只有矿工将一段时间内完成的真实交易封装并进行算力竞争，诚实节点构造的区块才能通过验证，并最终纳入区块链主链中，矿工也由此得到奖励的挖矿费。

如果矿工在构造区块的过程中有任何弄虚作假的行为，比如故意将比特币支付给自己的网络钱包等，由于哈希算法的特点，其他节点在验证时会发现其哈希值与父区块对应不上，因此直接丢弃这个区块，转而去验证别的节点发来的区块，那么作弊矿工构造的区块就不能通过验证，会被大多数节点所抛弃，也不能链入主链中，最终既得不到奖励的挖矿费，也浪费了本来可以通过算力竞争构造新区块的机会，还导致了电费、矿机损耗等方面的损失。在这种正向和反向双重激励之下，矿工诚实、勤勉地日夜工作，从而保证了比特币网络的稳定运行。

在新区块的验证通过后，矿工就以该区块为父区块，进行下一个区块的封装和计算。如果验证未能通过，该矿工的区块就会被抛弃。如果有其他的矿工广播出新的区块，那么网络将重复区块验证过程。如果同时有数个区块被计算出来，并且都可以通过全网的验证，那么其他矿工可以任选一个区块进行新区块的“挖矿”，并广播挖出的新区块。这时候就产生了“多链”现象。但由于网络节点随机性的存在，一般在不超过 6 个新区块的范围内，原来的“多链”就会出现长短不一的情况，此时“马太效应”显现，全网的节点都会选择最长的那条链进行后续的区块构造，其他的短链都被抛弃，从而保证了主区块链的唯一性。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

chapter FOUR

第 4 章

区块链共识机制

在区块链系统框架中，共识层提供了全网对交易和区块的共识，是直接在区块链中产生信任的方法和机制，也是区块链的核心要素。

到目前为止，我们已经知道，区块链所存在的分布式 P2P 网络系统没有中心化的机构来产生信任，各个节点是对等的，同时在交易过程中也并不需要对节点的细节进行详细了解。也就是说，交易过程并不需要节点双方产生信任，可以由互不信任的双方来完成交易，该交易是由全网节点来进行验证并形成信任的。

TIPS:

如果没有一种有效的共识机制，系统就缺乏了存在的基础，还得要回到中心化网络下，由中心化机构来提供和产生信任。所以，共识机制是区块链网络的生命线。

4.1 共识机制的意义

为了在全网节点之间达成共识，需要建立一种客观机制，不受人为的主观因素影响，来确保交易本身得到了全网的验证，以及确保区块的建立得到了全网的共识，并最终保证全网的区块链有且只有一条（总账账本只有唯一的一本，其他的都是副本），只有这样才能使全网达成共识，交易得到确认，区块链的不可篡改、可追溯等特点才能达成，整个区块链系统才能顺利地运转下去。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在私有链甚至联盟链中，系统内的各个节点通过线下确认机制建立起了信任，因为只是同一个组织内部的区块链，彼此之间都是可以信任的，因此不存在对共识机制的强烈要求。但对于公有链而言，节点之间并无信任基础，甚至连交易都是匿名的，因此共识机制是不可或缺的。

目前常用的共识机制有 3 种：Proof of Work（PoW，工作量证明）；Proof of Stake（PoS，权益证明）；Delegated Proof of Stake（DPoS，授权股权证明）。其中，工作量证明机制是比特币所用的共识机制，也是目前使用最广泛和成熟的共识机制；权益证明和授权股权证明这两种机制是针对工作量证明机制的不足之处进行优化和完善而产生的，其应用范围目前也集中在比特币之后产生的一些数字货币等。每种共识机制都有其优点，也有其不足之处，下面我们将逐一介绍这 3 种共识机制。

4.2 工作量证明机制

工作量证明机制的逻辑是：数字货币的产生需要付出一定的工作量和成本，不能凭空得来。

这种机制赋予了数字货币一定的商品属性，使得数字货币无须中心化机构的干预，市场自身可以通过“价格机制”对数字货币的供应进行自发调节。当数字货币价格上涨时，更多的人投入工作量证明来创造出更多的数字货币，增加了数字货币供应，使数字货币价格回落；当数字货币价格下跌到付出的工作量和成本之下时，创造数字货币的一部分人就会退出，减少了数字货币供应，使数字货币价格又会回升。这种机制保证了数字货币的价值稳定，使数字货币具有价值存储能力，从而使数字货币获得了人们的信任。

在人类历史上曾经使用贝壳作为货币，但并不是所有的贝壳都可以作为货币，比如普通的贝壳就不能作为货币，因为它满地都是，不费吹灰之力就可以得到，因此没有人会接受它。

仅供非商业用途或交流学习使用

TIPS:

只有通过选择符合要求的贝壳，然后进行打磨和钻孔，在符合一定的要求后，才能成为人们接受的货币。在这一过程中，选择贝壳和打磨贝壳的过程付出了一定的工作量，才使得贝壳币具备了信任的基础。

随着人类生产水平的提高，当贝壳币可以用极低的成本或工作量来大批量制造出来时，它就失去了“付出一定的成本”这个工作量证明机制的本义，不再被人们信任，也就不再成为货币了。

在比特币网络中就采用了工作量证明机制的原理，设计出一种“通过计算哈希数值的方式来证明完成了一定的工作量”的模式，来确保只有完成相当的工作量才能创造出比特币。

比特币采用的工作量证明机制是区块链技术中最早使用、目前也是最大规模使用的共识机制，被形象地称为“挖矿”。由于比特币作为第一种区块链数字货币不断吸引着投资者进入，使得比特币在全球范围内的价格持续攀升，进行工作量证明（挖矿）的收益维持在较高水平。在经济利益的推动下，世界各地都诞生了一些专门从事工作量证明的机构，并衍生出相关的产业链，被称为“矿池”，通过构建强大的算力来提高完成工作量证明的效率。

4.2.1　哈希函数

目前最常用的一种工作量证明机制计算方法是哈希散列法，它是由亚当·贝克（Adam Back）在 1997 年发明的，用于抵抗邮件的拒绝服务攻击及垃圾邮件网关滥用。在比特币诞生之前，哈希散列法被广泛用于垃圾邮件的过滤，并应用于比特币出现之前的试验性加密数字货币之中，如戴伟的 B-money、尼克·萨博的比特金（Bit-Gold）等，都应用了哈希散列工作量证明机制，所以这种方式也被中本聪借鉴用于比特币系统中。

哈希函数（Hash Function）也称为散列函数，给定一个输入 x，它会计算出相应的输出 $H(x)$。哈希函数有以下主要特征：①输入 x 可以是任意长度的字符串（这一点可以使哈希函数应用于对各种形式的数据进行加密）；②输出结果即 $H(x)$的长度是固定的；③计算 $H(x)$的过程是高效的；④安全性，也就

仅供非商业用途或交流学习使用

是说，对于一个给定的哈希函数输出结果 $H(x)$，想要逆推破解出输入值 x，在计算上是不可能的。

对于比特币这种加密系统所使用的哈希函数来说，在上述特点之外，还需要具备一个性质——免碰撞，即不会出现输入 $x \neq y$，但是 $H(x) = H(y)$的情形。因为如果容易出现碰撞现象，那么有可能出现两个用户的私钥相同，直接影响用户的比特币资产安全。

因此，比特币使用的哈希函数是更为复杂的 SHA256 算法，理论上有 2^{256} 种不重复的输出，只有当我们进行了（$2^{256}+1$）次输入时，才会必然发生一次碰撞（哈希结果重复）；或者当我们输入 2^{130} 次时，有 99%的可能会发生一次碰撞。

TIPS:

在实际情况中几乎不会发生碰撞。假设一台计算机以每秒 10 000 次的速度进行哈希运算，则需要 10^{27} 年才能完成 2^{128} 次哈希计算，这是一个极为漫长的时间，所以可以认为在现实条件下 SHA256 算法的结果几乎不会发生碰撞。

由于 SHA256 算法具有以上特点，所以成为比特币的工作量证明机制的核心算法。

4.2.2 工作量证明机制的基本原理

工作量证明机制的主要特征是请求方需要做一定难度的工作计算出一个结果，验证方却很容易通过验证结果是否满足要求来确认客户端是不是做了相应的工作。这种方案的一个核心特征是不对称性：工作对于请求方而言是有一定难度的，对于验证方而言则是易于验证的。

假如对一个给定的字符串“Hello, world!”，系统设定的工作量要求是：在这个字符串后面添加一个叫作 Nonce 的随机整数值，对变更后（添加了 Nonce）的字符串进行 SHA256 哈希运算，如果得到的哈希结果（以十六进制的形式表示）是以“0000”开头的，则完成了工作量，可以通过验证。

在验证开始之前，请求方并不知道到底哪个数值添加后可以达到前 4 位为“0000”的工作量证明目标，所以只有采用笨办法，即从 0 开始不断地递增变动

仅供非商业用途或交流学习使用

Nonce 值，并对每个变动后的新字符串进行 SHA256 哈希运算，再对照哈希运算结果与目标要求，直到找到那个随机数（Nonce）为止。通过下面的示例可以发现，需要经过 4251 次计算才能找到恰好前 4 位为 0 的哈希散列。

```
"Hello, world!0"=> 1312af178c253f84028d480a6adc1e25e81caa44c749ec819
76192e2ec934c64
"Hello, world!1"=> e9afc424b79e4f6ab42d99c81156d3a17228d6e1eef4139be
78e948a9332a7d8
"Hello, world!2"=> ae37343a357a8297591625e7134cbea22f5928be8ca2a32aa
475cf05fd4266b7
...
"Hello, world!4248"=> 6e110d98b388e77e9c6f042ac6b497cec46660deef75a
55ebc7cfdf65cc0b965
"Hello, world!4249"=> c004190b822f1669cac8dc37e761cb73652e7832fb81456
5702245cf26ebb9e6
"Hello, world!4250"=> 0000c3af42fc31103f1fdc0151fa747ff87349a4714df7
cc52ea464e12dcd4e9
```

当这个结果被计算出来后，请求方向验证方告知结果：在“Hello, world!”后面附加上 4250 可以使 SHA256 的前 4 位为“0000”。验证方无须再从 0 开始验证，它可以直接应用“Hello, world!4250”进行 SHA256 计算，发现请求方的计算结果确实满足了要求，于是验证得以通过。

4.2.3　比特币的工作量证明过程

在比特币网络中参与工作量证明的节点（矿工）实时监听网络，将接收到的广播交易信息存放到内存的交易池中，经过一段特定的时间（一般是 10 分钟），矿工将上述全部交易“组装”成区块中的交易信息，意图创建一个新的区块并写入区块链，此时必须解出比特币网络要求的工作量证明的谜题。这道题关键的 3 个要素是工作量证明函数、区块及难度值。工作量证明函数是这道题的计算方法，区块决定了这道题的输入数据，难度值决定了这道题所需的计算量。

1. 工作量证明函数

比特币使用的工作量证明函数是 SHA256。SHA 是安全散列算法（Secure Hash Algorithm）的缩写，是一个密码散列函数组。这组函数是由美国国家安全局（NSA）设计、美国国家标准与技术研究院（NIST）发布的，主要用于数字签名标准。SHA256 是这个 SHA 函数组中的一种算法，其输出值是 256 位的。

仅供非商业用途或交流学习使用

2. 区块

比特币的区块由区块头及该区块所包含的交易列表组成。区块头的大小为 80 字节，由 4 字节的版本号、32 字节的前一区块哈希值、32 字节的默克根、4 字节的时间戳（当前时间）、4 字节的当前难度值、4 字节的随机数（Nonce）组成。区块包含的交易列表则附加在区块头后面，其中的第一笔交易是对矿工进行比特币奖励的币基交易。

区块的大致结构如图 4.1 所示。

长度为 80 字节的区块头就是用于比特币工作量证明的输入字符串。为了使区块头能体现区块所包含的所有交易，在区块的构造过程中，需要将该区块要包含的交易列表通过默克树算法生成默克根，存入区块头数据中。

版本号	02000000
前一区块哈希值	17975b97c18edlf7e255adf297599b55330e dad87803c817010000000000000000
默克根	8a97295a2747b4fla0b3948df3990344c0e1 9fa6b2b92b3a19c8e6badc141787
时间戳	358b0553
当前难度值	535f0119
随机数(Nonce)	48750833
交易计数	63
币基交易	
交易	
.....	

图 4.1　比特币区块的结构示意图

3. 难度值

难度值（Difficulty）决定了矿工大约需要经过多少次哈希运算才能产生一个合法的区块。比特币的区块大约每 10 分钟生成一个，由于全网算力处于实时波动状态，如果要求新区块的产生都基本保持这个速率，那么难度值必须根据全网算力的变化进行调整。

难度值的调整是在每个完整的节点中独立自动发生的。每 2016 个区块，所有节点都会按统一的公式自动调整难度值，这个公式是由最近的 2016 个区块花费时长与期望时长进行比较，计算出实际时长与期望时长的比值，以此来相应调整目标要求的，如果新区块产生的速率比 10 分钟快则增加难度，比 10 分钟慢则降低难度。

这个公式可以总结为如下形式：

新难度值=旧难度值×(20160 分钟/过去 2016 个区块花费时长)

具体代码如下：

仅供非商业用途或交流学习使用


```
unsigned int CalculateNextWorkRequired(const CBlockIndex* pindexLast,
int64_t nFirstBlockTime, const Consensus::Params& params)
{
    if (params.fPowNoRetargeting)
        return pindexLast->nBits;

    // 限定调整步骤
    int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
    if (nActualTimespan < params.nPowTargetTimespan/4)
        nActualTimespan = params.nPowTargetTimespan/4;
    if (nActualTimespan > params.nPowTargetTimespan*4)
        nActualTimespan = params.nPowTargetTimespan*4;

    // 重定向
    const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
    arith_uint256 bnNew;
    bnNew.SetCompact(pindexLast->nBits);
    bnNew *= nActualTimespan;
    bnNew /= params.nPowTargetTimespan;

    if (bnNew > bnPowLimit)
        bnNew = bnPowLimit;

    return bnNew.GetCompact();
}
```

工作量证明需要有一个目标值，作为完成工作量的检验标准。比特币工作量证明的目标值（Target）的计算公式如下：

目标值 = 最大目标值 / 难度值

其中，最大目标值为一个恒定值，如下：

```
0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

TIPS:

由上述公式可知，目标值的大小与难度值成反比，即难度值越大，目标值越小。而比特币工作量证明的完成标准就是矿工计算出来的区块哈希值必须小于目标值。因此，难度值越大，目标值越小，矿工完成目标的时间就越长。

比特币工作量证明的过程就是通过不停地变换区块头及其附带的 Nonce 值

仅供非商业用途或交流学习使用

作为输入，进行 SHA256 哈希运算，找出一个特定哈希值的过程（要求该哈希值的前 n 位为 0，比如前 20 位为 0 的哈希值）。要求为 0 的位数越多，意味着难度越大。

一旦“解题”成功，矿工需要马上向其他节点进行广播，由其他节点进行验算，与当前网络的目标值进行对比（验证过程），如果确实小于当前的目标值，则验证通过，该区块创建成功，得到全网的认可；其他的矿工马上以这个区块作为父区块，继续“收集交易至交易池—组装区块—哈希运算”的挖矿过程……上述过程不断循环，完成了区块链的延长、交易的确认、比特币的产生，比特币系统得以正常运转。

4.2.4 工作量证明机制的优缺点

工作量证明机制的最大优点是相对公平，并解决了 P2P 网络中的信任难题，使共识得以达成，交易可以进行。

但同时工作量证明机制的缺点也非常明显。一是资源消耗增大。算力是由计算机硬件（从初期的 CPU 到后期的专用矿机）提供的，需要消耗大量的电力资源，这与当今世界节能环保的发展趋势形成冲突。同时这种挖矿计算并不产生实际的意义，除了确认交易和取得比特币，并不解决现实生活中的任何实际计算问题。为了挖矿成功，世界各地都有人投巨资建设矿场、矿池，购买大量采用 ASIC 芯片计算的矿机，但这些软硬件设施却只用于计算一个无意义的随机数（哈希散列），相当于造成了大量算力的浪费。

二是出现了中心化的趋势。以比特币网络为例，挖矿这种机制发展到现在，从个人计算机的 CPU、GPU（图形处理器，具有比 CPU 更强大的浮点计算能力）、FPGA 到现在采用 ASIC 芯片计算的矿机，在组织形式上已经发展到组建了大型矿池、矿场，配备了大量的专业矿机，可以更快速地完成工作量证明，从而取得比特币，结果就是全网的算力越来越集中，矿池成为中心化的计算机构。这与区块链网络去中心化的方向是背道而驰的，一方面背离了区块链的本质特点，另一方面在算力集中的情况下 51%攻击的可能性大大增加，比特币系统的安全性受到影响。

4.3　权益证明机制

权益证明机制[①]是针对工作量证明机制存在的不足而设计出来的一种改进型共识机制，权益证明的英文全称是 Proof of Stake，简称 PoS。与工作量证明机制要求节点不断进行哈希运算来验证交易有效性的机制不同，权益证明机制的原理是：要求用户证明自己拥有一定数量的数字货币的所有权，即“权益”，才能进行挖矿。

2012 年 8 月[②]，一位化名 Sunny King 的程序员推出了 Peercoin（PPC），采用工作量证明机制发行新币，采用权益证明机制维护网络安全。这是首次将权益证明机制引入加密数字货币领域，后来这种机制得到了市场的广泛认同。引入 PoS 机制的还有 BitShares、ShadowCash、Nxt、BlackCoin 等数字货币，以太坊（Ethereum）也计划通过硬分叉（Hard Fork）来从 PoW 机制转换到 PoS 机制。

4.3.1　PoS 机制与 PoW 机制的区别

PoS 机制与 PoW 机制的最大不同之处在于，只有持有数字货币的人才能进行挖矿，而且不需要大量的算力就可以挖到货币，避免了比特币网络中出现的“算力集中”趋势，回归到区块链“去中心化”的本质要求。

PoW 机制在产生区块时依赖算力，算力来源于矿机、矿场、矿池持续不断地进行哈希运算，需要消耗大量的电力，从而造成资源浪费。统计数据表明，按照目前的发展趋势，到 2020 年，整个比特币网络所消耗的电力将超过丹麦一个国家所消耗的电力，这说明比特币网络极不环保。

PoS 机制与 PoW 机制相反，PoS 机制在产生区块时不采用算力作为资源，而以对数字货币的拥有权益作为基础来产生共识。也就是说，创建区块的节点必须提供证明，让全网认同它在创建这个区块之前拥有一定数量的数字货币。

① 参考：https://en.wikipedia.org/wiki/Proof-of-stake。
② 参考：http://www.btc38.com/altcoin/ppc/2755.html。

TIPS:

PoS 机制的底层逻辑是：持有数字货币越多的节点，越有动力去维护数字货币网络的安全。因为一旦网络遭受攻击导致数字货币价格下跌，它们的利益损失更大。另外，如果外部攻击者需要发动 51%攻击，那么需要先持有 51%的权益，这就要花费大量的财力，而发动攻击后自己持有的这 51%权益的价格下跌，导致的损失远远高于 51%攻击的收益，因此就没有人愿意对 PoS 网络进行 51%攻击了。

PoS 机制的目标是：

（1）区块不可伪造。

（2）矿工的回报与其持币规模并不必然成正比例。

（3）创建新区块不需要过多的算力。

（4）单个成员不能控制整个区块链。

为了实现上述目标，PoS 机制设计了一些与 PoW 机制不同的共识规则。

（1）与比特币 PoW 机制下的币基交易（Coinbase Transaction）不同，PoS 机制的区块中的第一项交易必须为空交易，而且回报为 0 个数字货币（还记得在比特币 PoW 机制中是什么样的吗？区块中的第一项交易即币基交易必须是给矿工自己一定数量的比特币）。

TIPS:

为了奖励权益人，区块中的第二项交易必须是特定的“权益交易”，要求的格式为：至少有一个有效的输入（vin）；第一个输出（vout）必须为空的脚本；第二个输出（vout）必须不为空。

（2）权益交易必须遵守以下规则，才能使区块有效。

①第二个输出（vout）要么是公钥脚本，要么是存储了公钥数据、未花费的返回脚本。

②该交易的时间戳必须等于区块的时间戳。

③权益交易的总输出必须小于或等于（总输入+PoS 区块奖励+总的交易费用）。

④对应第一个输入的输出消费，至少须在 500 个区块之前确认（目的是不鼓励过分长时间地持有数字货币，而不参与对整个 PoS 区块链的维护）。

⑤在权益交易中虽然有多个输入可以使用和消费，但是第一个输入必须用作共识参数。

这些规则令权益交易很容易被识别，有利于 PoS 区块链对新区块实施快速确认。

（3）在 PoS 机制下对新区块有以下要求。

①必须有一个确定的权益交易。

②权益交易必须是区块中的第二项交易。

③币基交易（区块中的第一项交易）必须有一个输出值为 0，以及一个输出值为空。

④区块时间戳的前 4 位必须设为 0，这意味着区块的时间间隔为 16 秒，降低了颗粒度。

⑤区块的高度为 7（太早或太新的区块都会被拒绝）。

⑥区块的“核心哈希”必须满足 PoS 机制的加权难度要求。

⑦由存放在权益交易的第二个输出中的公钥对区块哈希进行数字签名；签名数据只存储在区块中，并不用于区块哈希过程。

⑧存储在区块中的数字签名由单一数据组成，并尽可能地压缩，不产生附加行距或其他操作码。

⑨区块的其他规则与 PoW 机制相同，如交易的默克根、交易验证等。

上述规则较多，但对于 PoS 机制来说最重要的是“核心哈希”，因为“核心哈希”与 PoW 机制相似，是作为区块创建的验证标准的，即只有在核心哈希值满足难度要求时区块才是有效的。但在当前区块中，“核心哈希”并不能直接被修改，其数据组成结构如下（按先后顺序）：

（1）前一个区块的“权益编译器”。

（2）前一个输出交易的时间戳（前一个输出交易是指权益交易的第一个输入所对应的前一个输出交易）。

（3）前一个输出交易的哈希值。

（4）前一个输出交易的编号（哪个输出被权益交易使用了）。

（5）当前区块时间，前 4 位为 0，这是权益交易阶段唯一变动的参数。

区块的“权益编译器（Stake Modifier）”是指一串哈希值，是对前一个输出交易和前一个区块的权益编译器进行哈希运算的结果。

TIPS:

> 因此，在 PoS 机制下创造区块的过程是不断地进行“核心哈希”的过程，而“核心哈希”的变动有两种方式：要么变动前一个输出，要么变动当前区块时间。

4.3.2 PoS 区块创建

对于数字货币来说，一个“钱包”（指私钥对应的钱包地址）一般包含许多未花费交易（UTXOs），这个钱包的资产就是所有未花费交易的总和，钱包可以支配这些资产。PoS 钱包也是这样的，这些未花费交易中的一个可能成为“前一个输出”，从而创建出 PoS 区块。

PoS 区块的挖矿流程还有一个变化，即难度值与权益交易的数字货币数量成反比。也就是说，如果在权益交易中投入两个币，那么与只投入一个币相比，难度值要减小两倍。所以，PoS 机制鼓励节点持有大量的数字货币，并用于权益交易，以降低区块创建的难度值。

用 Python 语言简要表示区块的创建流程如下：

```
while(true){
    foreach(utxo in wallet){
        blockTime = currentTime - currentTime % 16
        posDifficulty = difficulty * utxo.value
        hash = hash(previousStakeModifier << utxo.time << utxo.hash <<
utxo.n << blockTime)
        if(hash < posDifficulty){
            done
        }
    }
    wait 16s -- wait 16 seconds, until the block time can be changed
}
```

上述代码的意思是循环执行以下过程：

（1）计算区块时间（当前时间-当前时间与 16 的模数）。

（2）计算 PoS 的难度值，等于网络的难度值乘以钱包持有的数字货币个数。

（3）遍历钱包中的每项交易，用（前一个输出交易的权益编译器+交易自身数据+区块时间），计算其 SHA256 哈希值，并把这个哈希值与难度值相比，如果哈希值小于难度值，则说明“核心哈希”有效，可以创建区块。

如果所有的哈希值都不满足要求，那么再等 16 秒，等区块时间改变后，再次重复以上过程，直到哈希值满足要求。

（4）在完成哈希运算之后，把内存池中的所有交易写入区块，然后进行数字签名，这时要用到在权益交易的第二个输出中嵌入的公钥。在完成数字签名后，节点将区块向网络进行广播，其他节点验证通过后，完成区块的创建。

权益证明机制在创建区块时，要求节点先拥有数字货币，即先拥有一定数量的权益（Stake）。分配初始权益有两种做法：一种是在创世区块中把所有的数字货币发行出来，由各个节点购买已经发行的数字货币取得初始权益；另一种是从 0 开始到一定数量（比如 10 万个）的数字货币采用 PoW 方式挖矿完成，从一定数量之后的数字货币采用 PoS 方式挖矿完成。

4.3.3　PoS 机制的发展过程

PoS 机制自创立以来，也在不断发展，目前已经经历了 3 个版本的修订。

PoS 1.0 版，首先在 PeerCoin 中得到应用，采用了币龄（Coin Age，持有数字货币的时间）的概念，币龄越大则挖矿难度越低。这导致用户会选择间隔很长的时间（比如一个月或更长时间）上线连接钱包，这样钱包中的币龄都会比较大，用户可以很快地挖到新区块。由此影响到用户节点的参与程度较低，对网络的安全性不利，容易导致攻击产生。

PoS 2.0 版在共识机制中移除了币龄，并采用了和 PoS 1.0 版完全不同的锚定修饰符（Stake Modifier）。其他数字货币如 BlackCoin 使用了 PoS 2.0 版。

目前应用的是 PoS 3.0 版，它在锚定修饰符中移除了前一区块的币龄，可以防止“短距攻击（Short-range Attack）”。在 PoS 3.0 中 UTXO 年龄进一步简化，只由区块高度决定，可以避免“时间穿越（Timewarp）”攻击。在 PoS 3.0 中还增加了对锚定交易的 OP_RETURN 的支持，使交易输出中只包含公钥，而并不需要完整的 pay-to-pubkey 脚本。

TIPS:

更重要的是，PoS 3.0 版要求挖矿节点持续在线并积极参与权益交易，因此增加了在线节点共同维护网络交易的数量，提高了网络的安全性。

另外，持币少的人与持币多的人相比，只要全部时间在线参与权益交易，就可以得到比例更高的区块创建利息奖励。比如，持币多的人可能得到 1%的利息奖励，而持币少的人由于参与的时间长，可能得到 2%甚至更高比例的利息，从而激励人们多多在线。这种机制也使得交易费用的分配更加分散化，避免“富者更富，穷者更穷”的中心化趋势。

4.4 授权股权证明机制

授权股权证明的全称是 Delegated Proof of Stake，简称 DPoS，它是一种综合了 PoW 和 PoS 机制的优点、试图克服其不足的新型共识算法，由 Dan Larimar 于 2014 年 4 月 3 日在同名的白皮书中首次提出。这种算法的首次应用是在比特股（BitShares）上，后来由于其具有的优势也被其他数字货币或产品所采用，如 Crypti、Lisk 等都采用了 DPoS 算法。

TIPS:

DPoS 虽然名称简写仅比 PoS 机制多出了一个 D 字，但是 DPoS 与 PoS 相比还是有较大的差异的，最重要的一点是：铸币权不再由系统根据特定算法来随机选出，而由网络的全部节点基于权益随机投票选出 101 名授权委托代表（Delegate），由这 101 名代表进行铸币及维护网络安全。

4.4.1　DPoS 概述

在 PoS 机制下，每个持有数字货币的钱包（账户）拥有权益，可以确认交易、参与铸币及赢得数字货币。但在 DPoS 机制下，每个持有数字货币的账户（我们称之为“股东”）可以投票选出授权代表，由这 101 名代表（注：在最初的比特股系统中设置的是 101 名授权代表）来完成确认交易、维护区块链及获取交易费用的操作。

比特股自创立以来也在不断改进和优化，目前应用的是 BitShares 2.0 版本，在原有的 DPoS 机制上进行了优化，除原来的授权代表继续存在外，又开创性地建立了“见证人（Witnesses）”机制，即全部节点还要再投票选出见证人，见证人节点专注于进行交易确认、区块创建的工作，并取得一定的报酬；授权代表节点则不再进行区块创建，专注于网络协议的维护，以及在必要时进行更新。

但是，授权代表提出的对网络及协议的修改方案并不能直接执行，必须经过全部股东的“全民公决”来决定是否同意，如未获得股东认可也不能执行。

所以在 BitShares 2.0 版本中，这个机制很像一家公司：股东大会拥有最高决策权；授权代表类似于公司的董事会，制定重大规则，提交股东大会审核通过后执行；见证人节点类似于公司的管理层，进行日常经营行为（验证和确认交易、创建区块）。

4.4.2　选举见证人

DPoS 机制中设计的选举系统确保了股东对于授权代表的控制，避免了让创建区块的权力完全集中在少数人的手里，增强了系统的去中心化特点。比如，某个股东持有 10 个代币，那么他可以给 101 名授权代表和见证人每人投出 10 个代币的权重。这一点与 PoS 机制有所不同，在 PoS 机制下，持币越多越有利于创建新区块，容易导致“贫富分化”；但是在 DPoS 机制下，每个节点（股东）的持币数量并不直接对挖矿（区块创建）产生影响，而是间接地通过投票选举授权代表和见证人完成区块的创建，共同维护网络运行，这就避免了 PoS 机制的不足之处。



TIPS:

DPoS 机制下的每个用户可以决定谁受到信任，而不是只能将信任托付于那些掌握了绝大部分资源的人（比如 PoW 机制下的大矿场、PoS 机制下的持币大户等），所以 DPoS 机制虽然看上去有一点中心化，但其实它的中心化程度是受到控制的，通过牺牲一定的去中心化程度换来了效率的大幅提升。

见证人的数量由股东决定，可以为 50 名，或者 20 名，但在 BitShares 2.0 版本中规定，至少要有 11 名见证人，以确保网络的安全运行。投票方式采用的是“认可投票制”，有利于降低持币数量多的股东的影响力，提高去中心化程度。

见证人在成功创建区块后可以得到奖励，奖励标准由授权代表决定；但是如果做得不好，就很容易失去位置，做不成见证人，因为系统只会给前 20 名见证人奖励。在网络中同时有数百名见证人作为备选，而且投票是不断进行的，一旦某个见证人不好好干活，股东可以很容易地在下一轮投票中把他换掉。

每过 1 天（网络的维护间隔期），活跃见证人的候选名单就会更新一次，对见证人进行洗牌，去掉不合格的见证人，换进来新的见证人。在创建区块的过程中，首先由上一个区块随机产生下一个区块的创建者（见证人的顺序号），然后由该名见证人在 2 秒内进行区块创建工作，如果未能完成创建，那么系统自动切换到下一名见证人来进行下一个区块的创建，跳过的见证人就不能得到区块创建奖励。当所有的见证人全部轮完后，会再次洗牌。

4.4.3 选举授权代表

授权代表的选举方式与见证人的选举方式相似，但授权代表只负责维护网络、提出修改建议，比如区块大小、见证人的酬劳、交易费率等。

每位授权代表都有一个特殊账户，有权提出对网络参数的修改建议，这个账户被称为创世账户。当大部分授权代表都同意提交修改建议后，股东有两周的时间对修改建议进行审核，期间股东可以投掉或者废除授权代表的提议。这种机制使得授权代表没有直接权力，而且网络参数的变更最终是由所有股东来决定的。在 DPoS 机制下，可以说网络由用户掌控，而不是由授权代表或见证人来掌控的。



授权代表没有奖励，但是网络参数也不是经常变动的，因此授权代表的工作量并不大。

DPoS 机制的区块中装入的交易数量超过了 PoW 机制（如比特币）和 PoS 机制的区块中装入的交易数量，而且它的交易确认时间可以与现实生活中的中心化交易模式（如万事达信用卡的清算中心等）相媲美。

由于采用了 DPoS 机制，所以比特股的运行参数比绝大多数区块链应用产品的运行参数要优秀得多。

（1）区块时间：平均 1.5 秒（比特币为 10 分钟），最大 3 秒，99%以上为 1.5 秒。

（2）区块奖励：1 个 BTS（比特股的代币）。

（3）不可撤销区块数：34 个。

（4）每秒可处理的最大交易数量：具备 10 万笔以上的处理能力，目前已经证实的在 3400 笔以上。如能实现每秒 10 万笔以上的交易处理量，就已经超过了全世界信用卡交易的总量。

4.5　区块链的三大类型

随着区块链技术的快速发展，越来越多的政府机构、各类企业及开发人员参与其中，各种各样的相关应用和产品先后被开发出来并上线运行，但目前出现的区块链技术应用和产品有不同的应用范围和技术特点。根据不同区块链产品的底层架构和应用范围，人们一般把区块链分为 3 种类型：公有链、私有链及联盟链。

4.5.1　公有链

公有链[①]是参与程度最广泛的区块链，也是区块链技术得以附着的底层基

① 参考：http://www.guopeiwang.com/Article/ArticleDetial/20718。

础应用。每款区块链应用产品，如果从软件架构的角度向下“穿透”，那么一定会最终到达某条公有链。从严格意义上来说，公有链是指全世界任何参与节点都可读取的、任何参与人都能发送交易且交易能获得有效确认的、任何人都能参与其共识过程的一种开放的区块链，甚至连源代码也必须是开放的，因为它需要向位于其上的各个区块链典型应用开放接口。公有链通常被认为是“完全去中心化”的，在理论上由全部节点参与的共识过程来决定区块链主链的区块创建，以及在面对冲突时选择哪条最长的子链作为主链等。公有链是区块链世界的逻辑基础，现有的互联网络是区块链世界的物理基础。

公有链具有以下 3 个特点。

（1）用户与开发者隔离，使用户免受开发者的影响。鉴于公有链用户的广泛参与程度，必须最大限度地保证用户与程序开发者的相互隔离，使用户的各种应用不会受到开发者的影响。通过开放源代码、提供不同的接口函数，由开发者进行调用，可以实现功能封装和调用，保证公有链的稳定运行。

（2）较低的使用门槛。要达到广泛的用户参与，就需要降低使用的门槛，因为世界上不同地方不同的节点具备的基础条件完全不一样，如果门槛过高，则必然会限制一部分人的访问，无法达到公有链的参与程度。一般来说，只要能够上网，就能够访问公有链，这样的区块链才具备成为公有链的条件。

（3）全部区块链数据处于公开状态。这是由区块链的去中心化特性所决定的。首先，作为一本分布式的总账，各个节点都会保存区块主链的备份并对其加以确认；其次，对于已经存在于区块链上的区块，其内部封装的交易信息在理论上可以由任何一个节点通过遍历访问，一直可以追溯到创世区块，只有这样才能避免被恶意篡改。虽然账本是公开的，但是用户的身份信息通过私钥加密，只能看到一串复杂的字符串，并不知道用户的真实情况，所以对于用户而言是一种比较保密的状态，这一点也得到了许多人的认可。

TIPS:

目前最成功、最广为人知和被全世界公认的公有链就是比特币网络。此外，以太坊、其他一些数字货币等也具备公有链的大部分特点和应用范围，也可以被视为公有链。由于以太坊与比特币相比具有更好的应用开发



特性，更加开放，而比特币网络比较封闭，性能不能满足商业实践的需求，因此以太坊的接受程度越来越高，成为应用开发者最认可的公有链。

在公有链建立以后，主要功能就是利用其数据公开、无法被篡改的特性，实施全网共同“验证”、智能合约等应用功能，将数据最终封装到区块链中。以“验证”为例，在比特币区块链上，由于在每个区块的产生过程中都会对大量的信息进行哈希运算，并进行默克树计算，因此，只要将某些证明信息封装到区块中，借助公有链的不可篡改特点，就可以保证相应的证明信息具备足够强大的安全性和证明力。

例如，比特币中国（BTCC）曾经推出“千年之链”区块链刻字业务，在利用自营矿池的算力取得创建区块的权利后，在封装比特币网络的交易信息过程中，也可以同时接受客户将自己想写的信息加入交易信息中，并封装到区块中、链接到主链上，成为后续无数新区块的起点。只要比特币区块链稳定运行，这些刻上去的“字”就会由全网的分布式节点进行维护，任何人都难以更改。

公有链的不足之处也是很明显的。一是交易确认速度慢。由于参与的用户众多，需要大量的用户来形成共识，以便对新的区块构造权进行竞争，所以需要采用一些规则（如 PoW、PoS、DPoS 等）来达成共识，使绝大部分节点能够对一段时间内发生的交易进行共同验证和确认，这就需要花费较长的时间，从而降低了交易确认的速度。二是参与的用户分布广泛。虽然安全性得到了极大的增强，但是对于一些需要保持隐秘的交易，如果被封装到区块链上，则可以被任意节点访问。这样一来，虽然节点的具体身份可以保持隐秘，但交易信息却能够被大量节点看到，要求的保密性也就丧失了。

TIPS:

此外，公有链的交易量巨大，虽然采取了默克树算法来减少数据量，但仍然会导致公有链的分布式账本数据容量很大。以比特币为例，自创建至今的全部区块完整交易记录的数据量达到 145GB 以上。如果一个接入公有链的节点想要保存和更新这一分布式账本，则对于存储容量、网络带宽都有不低的要求。



4.5.2 私有链

与公有链有较大不同的区块链称为私有链。从字面意思上来理解，私有链相对于公有链来说，其应用的范围大幅缩小，不再是任意某个人都可以访问的区块链，而是只有某些特定的节点才能访问和应用的区块链，具备了“私有”的特性。

在定义上，私有链是指区块链的创建及维护权限由一个组织拥有，非组织成员仅拥有访问权限或无法访问的小规模区块链网络。

私有链的特点如下：

（1）用户规模小，一般属于同一组织，交易速度快。由于每个节点属于同一组织，彼此之间已经完全建立了信任，所以无须复杂的共识机制即可进行交易验证，从而大幅提高了交易速度。

TIPS:

> 由于节点之间存在高度的信任，所以一条私有链的交易速度远远高于所有公有链的交易速度，甚至可以接近常规数据库的存取速度。

（2）信息安全性大幅提高。在私有链中，组织可以使用与其他区块链完全不同的数据加密算法，使自己这个系统内的数据仅能够由本区块链的各个节点识别和使用，离开了本区块链，其他的节点无法识别或使用，因此其安全性大大提高。

（3）交易成本大幅降低。由于各个节点同属于一个组织，彼此之间可以高度信任，不再需要花费资源来验证交易，因此在私有链上可以进行完全免费的交易。即便一项交易需要验证或确认，但由于私有链的规模小，所以信息交流和传输也可以很快完成，速度远远高于一般的公有链。

（4）安全性能大幅提升。因为私有链是相对封闭的，陌生节点无法接入，并且节点之间互相了解，所以公有链的各种恶意攻击现象不太可能发生在私有链上。

由于私有链的安全性能强，使得银行等金融机构也产生了对区块链技术的强烈兴趣，并纷纷加入进来，开发自己的私有链应用。

目前在全球范围内，有 Linux 基金会、R3CEV Corda 平台及 Gem Health 网络的超级账本项目（Hyperledger Project）等几个私有链项目。其中，R3CEV 是一家总部位于纽约的区块链创业公司，由其发起的 R3 区块链联盟至今已经吸引了多家大型银行的参与，包括富国银行、美国银行、纽约梅隆银行、花旗银行等。



TIPS:

有些人认为，私有链一般由某个企业或组织使用，并且由这个企业或组织为私有链上的用户提供信用背书，这实际上是一种中心化的存在。另外，也有人认为私有链其实不是区块链，而是一种分布式账本技术。

总的来说，私有链应用于传统金融机构，可以在一定范围内有效地解决传统金融机构的效率、安全和欺诈问题，但它并不会对金融系统的运行规则产生颠覆性的影响；而公有链则具备去中心化的力量，它对传统金融机构的大多数功能具备颠覆性的改变，可以从根本上改变金融体系的运作方式。

4.5.3　联盟链

联盟链是由多个组织或机构通过联盟的形式组建的区块链，联盟参与者之间通过契约或其他形式建立了信任和共识机制，构造区块和上链功能仅限于联盟参与者拥有，访问权限可以对外采取限制性开放原则。

1. 联盟链的特点

联盟链从参与规模上来说，是介于公有链和私有链之间的一种区块链，但联盟链与私有链之间、联盟链与公有链之间的规模界限并不十分严格。比如R3CEV，最初定位于部分银行之间的私有链开发，但随着全球加入的银行和其他金融机构数量不断增加，其参与范围也迅速扩大，事实上已经成为多家金融机构联合开发的联盟链。

联盟链具备“去中心化”的特点，它相当于在某个有限的范围内构造了一种“公有链”，比如金融机构之间，或者某个行业的全体参与者之间；在建立联盟链后，联盟的参与者按区块链的共识机制进行交易的验证和确认，不再需要外部的中心化信任机构，可以在一定程度上减少流程的损耗，提高交易速度和效率。所以，从交易速度上来看，联盟链的交易速度一定快于公有链的交易速度，但相对于更加封闭、信任程度更高的私有链，联盟链的交易速度就慢得多。

联盟链的存在往往意味着一种与公有链不同的区块链规则，这样才能保证联盟链的内部安全性。但是联盟链与公有链之间一般都会留出接口，使公有链可以在有限度的权限内访问联盟链；反过来，也可以把联盟链的信息经过哈希



运算后把最终的默克根链接到公有链上，借助公有链的公开和透明机制来保证联盟链的健壮和不可篡改。

TIPS:

简单来说，联盟链可以理解为“缩小规模的公有链”或“扩大规模的私有链”。由于它具备了区块链技术的绝大部分特点，同时又具有更安全、更快速、更接近商业实践的特点，目前处于快速增长的状态，很多“区块链+传统产业”的应用其实就是一种联盟链。

2. 联盟链应用：京东区块链

2017 年 6 月 8 日，京东商城携手 27 家品牌商成立了“京东品质溯源防伪联盟”，运用区块链技术搭建了“京东区块链防伪追溯开放平台”，通过联盟链的方式，实现线上线下零售商品的追溯与防伪，将更有效地保护品牌和消费者的权益，帮助消费者持续提升在京东的品质购物体验。

京东认为，伴随着中国政府将供应链创新与应用上升为国家战略和居民消费的不断升级，供应链风险控制和供应链透明度提升的诉求不断攀升。区块链技术可以搭建供应链全流程节点共同维护的联盟链，在联盟链中建立数据维护的参与规则和激励机制，鼓励供应链节点中的企业参与和维护供应链数据，促进供应链数据的协同和互通，进而提升整条供应链的透明度，同时也可为消费者购买商品的溯源和防伪提供技术支持。为此，京东积极筹备开放支撑自身落地应用的区块链 BaaS 平台，帮助政府、物流商、品牌商、金融机构等合作伙伴组建适用的区块链技术平台。京东的目标是以区块链为“链接器”，结合自身在云计算、大数据、人工智能、物联网等新技术上积累的经验，构建一体化的智慧供应链体系、零售网络和金融科技，拉近商品与客户的距离。

从信息技术上来看[①]，供应链由众多参与主体构成，存在大量交互协作，信息被离散地保存在各自环节、各自系统中，缺乏透明度。信息的不流畅导致各参与主体难以准确了解相关事项的实时状况及存在的问题，影响供应链协同效率。当各主体间出现纠纷时，举证和追责耗时费力。未来企业市场范围越来越大，物流环节表现出多区域、长时间跨度的特征，需要智能、高效的防伪追溯能力。区

① 参考《京东区块链技术白皮书》。



块链技术通过提供完整流畅的信息流、不可篡改的签名认证机制，可以实现去中心化或多中心化的精准追溯和充分信任，天然地适用于供应链管理。

借助区块链技术，实现品牌商、渠道商、零售商、消费者、监管部门、第三方检测机构之间的信任共享，全面提升品牌、效率、体验、监管和供应链整体收益。将商品原材料采购过程、生产过程、流通过程、营销过程的信息进行整合并写入区块链，实现精细到一物一码的全流程正品追溯。

具体技术方案的实现路径是：每条信息都拥有自己特有的区块链 ID“身份证”，且每条信息都附有各主体的数字签名和写入时间戳，供消费者查询和校验。区块链的数据签名和加密技术让全链路信息实现了防篡改、标准统一和高效率交换。

3. 京东区块链商业模式

京东区块链的目标是提供企业级的区块链技术与服务，建设具有高性能、良好扩展性、广泛场景通用性、安全合规、接口友好和易部署管理的区块链基础网络设施，打造开放共赢的区块链技术与服务生态。

京东区块链支持以下几种部署模式：

（1）参与主体维护完整的共识节点，好处是参与者可持有数据，但需要付出运维成本。

（2）参与主体仅维护网关节点，通过公共的共识节点接入区块链，好处是便于自行管理私钥，且维护成本低，但网关节点不持有数据。

（3）参与主体通过公共的网关节点接入，适用于 2C 场景，用户可以基于公共的网关节点托管私钥。

（4）监管方可以只部署“备份节点”，从其他节点处同步数据作为备案。

（5）监管方还可以部署共识和网关节点，对公众开放查询，作为“存证公示”。

如图4.2所示为京东区块链的部署架构(来源:《京东区块链技术白皮书》)。



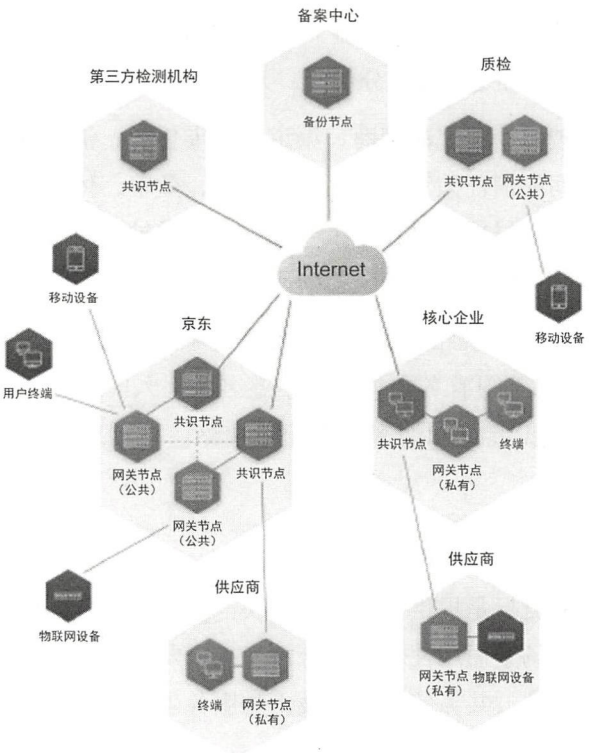


图 4.2　京东区块链的部署架构



chapter FIVE

第5章

区块链数据结构

从技术实现的原理和过程来看，区块链技术是利用区块式数据结构来验证与存储数据，在分布式网络的各个节点上运行共识算法由全网来维护和更新数据，利用密码学的方式保证数据传输和访问的安全，在此基础上采用由自动化脚本代码组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算体系。我们可以把它想象为一个一个区块以单向链接形式排列在一起的数据库，而且分散存储在网络的各个节点上。由于区块之间是简单的单向链接，因此区块内部的数据结构基本上包涵了区块链技术的主要内容，重要性不言而喻。

在区块链技术的各种应用中，区块包含并确认待处理的交易，是交易数据的集合；它还包含由系统时间产生的时间戳，并具有前一区块的标记。一个有效的区块只有在得到全网的共识后，才能被添加到区块链中，并且一旦进入区块链就成为永久的记录（不可篡改）。

本章将深入数据与算法层面，为大家详细介绍支撑区块链应用的底层技术。

5.1 区块链的数据组成

区块链的最大特点是去中心化。在去中心化的网络中，信息的交流和信任的传递都要依靠区块及其数据结构来实现。要了解这一点，就需要深入区块之中。

目前最成熟的区块链应用是比特币，所以我们以比特币的区块链技术为基础，对其中的数据结构进行探索。

5.1.1 区块的数据结构

区块是区块链网络的基本组成单元。区块不断地由各个矿工节点创建出来，并链接到区块链上，从而构成了区块链数据库，一旦链接成功就很难被修改或移动。

每个区块的数据结构如表 5.1 所示。

表 5.1 每个区块的数据结构

大　小	字　段	描　述
4 字节（固定不变）	神奇数	总是 0xD9B4BEF9，用作区块的分隔
4 字节	区块大小	用字节表示（该字段之后）的区块体积
80 字节	区块头	区块头的组成字段
1～9（可变整数）	交易计数器	记录交易的数量
变量	交易信息	记录进入区块里的交易信息

在这个数据结构中，能够与前一区块联系和链接起来的信息存放在 80 字节的“区块头”数据结构中，它由 3 组“区块头元数据”组成：首先是一组引用前一区块哈希值的元数据，通过这组元数据可以链接到前一区块，各个区块通过这一链接形成单向的区块链组，从任意一个区块向前回溯，最终可以回溯到创世区块；其次是一组元数据，表示难度目标、时间戳和随机数（Nonce），代表本区块生成过程的信息；最后一组元数据是默克根，用来有效汇总区块链中所有交易的数据信息。

区块头的数据结构如表 5.2 所示。

表 5.2 区块头的数据结构

大　小	字　段	描　述
4 字节	版本号	用于跟踪软件或协议的更新
32 字节	父区块哈希值	引用前一区块的哈希值
32 字节	默克根	本区块中所有交易的默克根的哈希值
4 字节	时间戳	该区块产生的近似时间（精确到秒）

续表

大　小	字　段	描　述
4 字节	难度目标	本区块工作量算法的难度目标
4 字节	Nonce（随机数）	用于工作量证明算法的计数器

5.1.2　区块链数据结构的技术基础

从上一小节中可以看出，区块链的数据结构是由一些基础技术生成的元数据组成的。在比特币的区块头中直接应用的是元数据的结果，那么这些元数据又是怎样得来的呢？

区块链的数据结构是在 SHA256 算法的支持下实现的；而在矿工进行交易封装、创建区块的过程中，采用高效的默克树算法一步一步地将繁杂的信息进行归纳和简化；同时，为了保证区块的顺序链接，必须采用“时间戳（Time Stamp）”机制在区块中写入时间参数；为了保证共识机制的达成，使全网参与的“挖矿”工作顺利进行，系统在共识机制 PoW 下采用了难度目标（Difficulty Target）和随机数（Nonce）两种设计，保证系统在一种稳定的状态下（每 10 分钟产生 1 个区块）运转。这些基础性数学或密码学技术产生了区块的元数据，它们是区块链数据结构的技术基础，如图 5.1 所示。

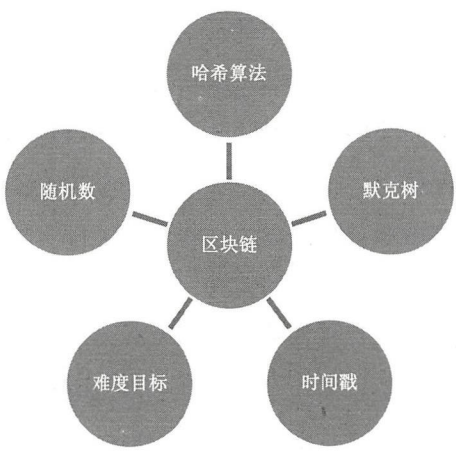


图 5.1　区块链数据结构的技术基础

下面对其中的各个基础模块做进一步的介绍。

5.1.3 哈希算法

哈希函数，或散列函数，又称为哈希算法，是一种从任何一种数据中创建小的数字“指纹”的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。所有的散列函数都有一个基本特性：如果两个散列值是不相同的(根据同一函数)，那么这两个散列值的原始输入也是不相同的。这个特性使散列函数具有确定性的结果，具有这种性质的散列函数称为单向散列函数。

哈希算法将任意长度的二进制值映射为较短的、固定长度的二进制值，这个固定长度的二进制值称为哈希值。哈希值是一段数据唯一且极其紧凑的数值表示形式。

TIPS:

如果对一段数据采用哈希算法处理后，哪怕只更改该段落中的一个字母，随后的哈希计算都将产生不同的值；但是反过来，对于哈希值相同的两个数，通过穷举的方式想要找到两个不同的输入信息(称为“反向碰撞”)，对于随机性能优秀的散列函数来说，这种方式在计算上是不可能的，所以数据的哈希值可以用来检验数据的完整性。

散列函数的这种单向特点和输出数据长度固定的特征使得它可以生成消息或者数据。基于这一特点，哈希算法一般用于快速查找和加密算法。典型的哈希算法包括 MD2、MD4、MD5 和 SHA-1 等，在比特币系统中，采用了随机性能更优秀的 SHA256 算法。

根据设定的散列函数和处理冲突方法将一组关键字映射到一个有限的地址区间上，并以关键字在地址区间中的镜像记录在表中的存储位置，这种表称为哈希表或散列，所得存储位置称为哈希地址或散列地址。

在区块链技术中，哈希算法在本质上是一个压缩数据的过程，即从原始的任意长度的数据（对应的可能是文档、图片、视频、音频等数据文件）开始，经过哈希计算后，压缩成固定长度的二进制值。而且由于其具有单向特性，因而可以通过哈希值对比来验证原来的数据是否发生变动。应用在区块中，既减

仅供非商业用途或交流学习使用

少了区块头存储的数据量，也相当于将整个区块的交易纳入区块的计算过程中，使恶意篡改变得无比困难，保证了区块链的数据安全性，因而成为区块链中的一项重要基础技术。

哈希算法在编程开发中通过调用散列函数即可实现。如利用 Python 标准库中的 hashlib 模块，可以完成对任意字符串的哈希计算，代码如下：

```
import hashlib
hash_object = hashlib.sha256(b 'Hello World')  #在此处可输入任意字符串
hex_dig = hash_object.hexdigest()
print (hex_dig)
```

5.1.4　默克树

默克树是一种用来保持大型数据库中数据一致性的高效方法，其采用了哈希算法来实现这种功能。具体来说，就是在比较网络中两个节点的数据是否完全一致时，不需要耗费大量的资源去比较全部的数据，只需要简单地比较存在差异的部分，对于完全一致的内容不再进行比较。如果网络采用了默克树算法，则可以很快地找到差异所在。

下面通过一个简单的例子来说明默克树的高效。

假设有两个节点：节点 1 和节点 2，它们保存的数据如下。

节点 1：1,2,3,4,5,7,8,9,10,11（缺失 6）。

节点 2：1,2,3,4,5,6,7,8,9,10（缺失 11）。

要比较这两个节点的数据一致性，并找出缺失的数据，在采用默克树算法时，第一步是把每个节点的数据分为 4 个分区，每个分区存储 5 个数据，如图 5.2 所示（黑色的部分表示每个节点缺失的数据）。

仅供非商业用途或交流学习使用

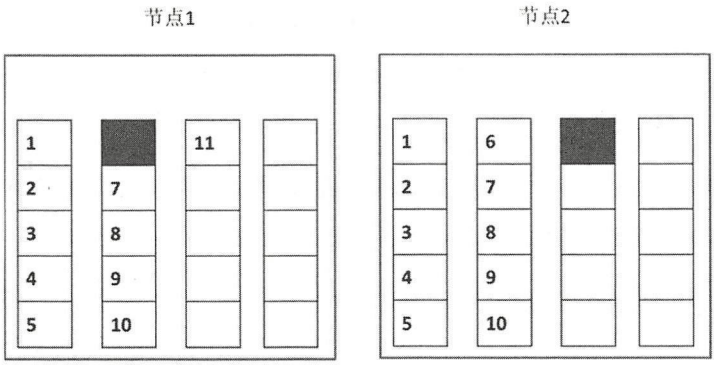


图 5.2　两个节点数据示意图

第二步是对每个存储的数据采用相同的哈希算法进行计算，得出其哈希值，缺失的数据无哈希值。结果如图 5.3 所示。

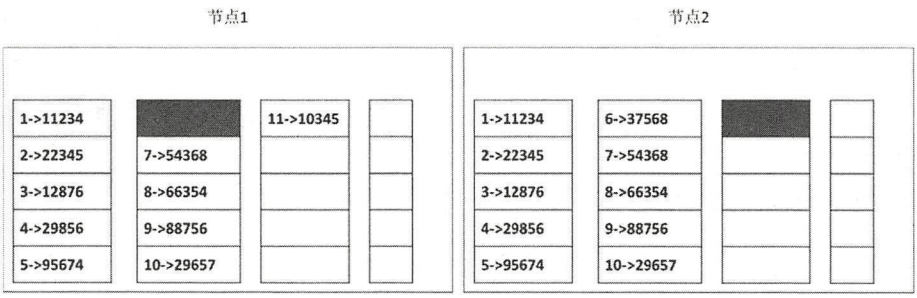


图 5.3　存储数据哈希计算结果

第三步是计算每个分区的哈希值。由于哈希算法具有单向性、唯一性，所以相同的数据得到的哈希值相同，有差异的数据得到的哈希值不相同。得到的结果如图 5.4 所示。

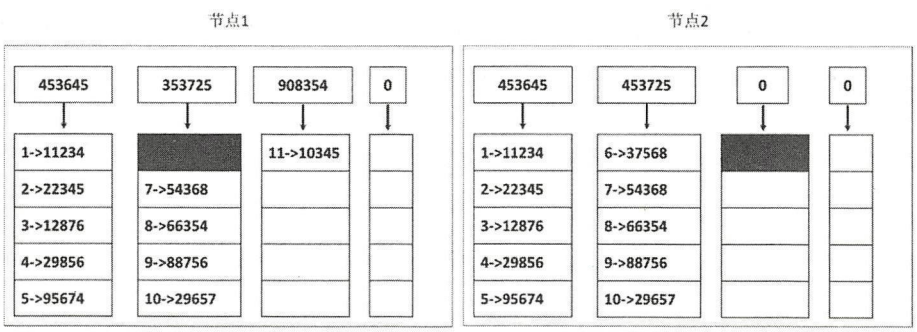


图 5.4　计算分区哈希结果

仅供非商业用途或交流学习使用

第四步是把每两个分区结合到一起，作为默克树的“叶子”，向上计算其哈希值。过程如图 5.5 所示。

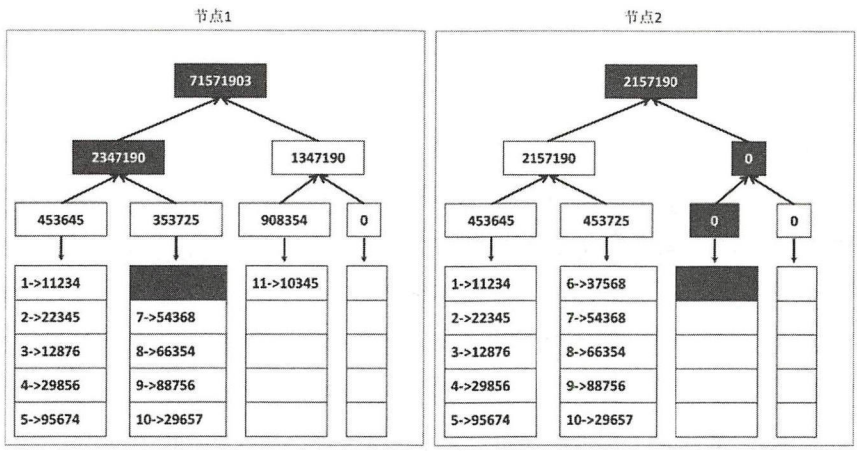


图 5.5　默克树向上哈希

完成以上步骤的过程就是把每两个节点的数据用默克树算法进行处理，最后得到的哈希值就是默克根。

此后，在对这两个节点进行数据同一性校验时，不需要对比全部数据存储区域的每个数据，而只需要从默克根开始校验，即：

（1）对比两个节点的默克根。如果完全一致，则说明全部数据是统一的，直接同步即可。如果两个节点的默克根不相同，如同本例中图示的情况，则说明这两个节点的数据存在差异，需要进一步比较。

（2）各个节点从根节点开始向下访问子节点并再次对比哈希值，节点 1 左侧的子哈希值（2347190）与节点 2 左侧的子哈希值（2157190）不同，说明左侧子节点的数据存在差异，需要进一步向下延伸比较。同理，右侧的子节点哈希值也不相同，说明右侧子节点的数据也存在差异，需要进一步向下延伸比较。

（3）访问孙节点的哈希值并进行校验。可以发现，第一组孙节点的哈希值相同（均为 453645），说明这一组中的数据完全相同，因此本组数据不需要进行对比。再比较第二组孙节点的哈希值，存在差异，说明数据出现了差异，需要对这一组中的数据进行详细的校验。此时将两块区域的 5 个数据进行对比，

仅供非商业用途或交流学习使用

可以发现节点 1 第二组的数据缺失，于是系统可以用节点 2 的数据对节点 1 的对应区域进行同步写入。

（4）对节点 1 和节点 2 右侧的数据重复上述过程，直到完成全部数据的校验和更新，使节点 1 与节点 2 的数据实现了完全一致。从模拟过程中可以看出，这种采用默克根进行比较和检验的过程相对于把整个数据库中的数据校验了一遍，但前者显然更为高效。

在区块链的交易中，每个区块记录了大量的交易，为了提高效率，减少资源耗费，将这些交易的哈希值两两进行哈希运算，直到形成最后一个哈希值。也就是重复上述的默克树算法，最后得到的一个哈希值就是默克根，这个默克根会被写入区块头中，只有 32 字节大小，易于传播，并且在验证时只需验证默克根即可，从而确保了数据的一致性。

使用 Python 语言对上述过程表述如下：

```
import hashlib
import binascii
coinbase_hash_bin = hashlib.sha256(hashlib.sha256(binascii.unhexlify
(coinbase)).digest()).digest()
def build_merkle_root(self, merkle_branch, coinbase_hash_bin):
        merkle_root = coinbase_hash_bin
        for h in self.merkle_branch:
                merkle_root = doublesha(merkle_root + binascii.unhexlify(h))
        return binascii.hexlify(merkle_root)
```

5.1.5 时间戳

时间戳是由时间戳服务器为每个区块加上的时间序列，记录了该区块的产生时间。时间戳采用了 UNIX 的时间计数方式，一般会精确到秒，即记录从 1970 年 1 月 1 日零时到当前区块产生时间的秒的累计数。

有了时间戳的存在，区块上的交易信息便无法更改，而且可以用作交易的证明。因为时间戳是写入区块头上的，同时在计算哈希值的过程中会将父区块的时间戳纳入，从而形成了对前一个时间戳的“增强”，而区块一旦被链接上区块链，就成为全网所有节点的“公共账本”，很难被篡改，从而增强了系统抵抗恶意攻击的性能。

仅供非商业用途或交流学习使用

5.1.6　难度目标

区块链是由全网络共同参与维护的，全网各个节点要达成一致，需要依靠共识机制。在比特币网络中[①]，共识机制是工作量证明机制，这是一种可以让每个参与的节点共同参与交易验证的方式，来实现一本多方共同维护并共享同一份记录交易的账本，在无信任处创造出信任来。

工作量证明机制通过让参与的节点花费时间和运算资源计算出一组数学计算的结果来完成区块的创建过程，一旦这个数值被算出来后，就会广播给其他参与节点进行验证。验证的标准即难度目标，只有计算结果小于难度目标值才能通过其他节点的验证。

进行 PoW 计算的过程也被形象地称作“挖矿”，其过程如下。

（1）每个区块的区块头中会包含不同的元数据，大部分是固定值，只有随机数（Nonce）是随机变动的。因此，每个节点在进行 PoW 计算时，通过不断替换这个 Nonce 值，来让这个区块的区块头哈希值小于一个被设定好的难度目标值。

TIPS:

为什么要小于这个难度目标值？因为难度目标本身设定得非常小，哈希计算结果通常是大于这个难度目标值的，矿工只有通过不断地改变随机值，进行哈希碰撞，才能使计算结果比难度目标值小，这样才能从机制上保证是经过了大量计算才完成挖矿的。反之，假设规定“大于难度目标值”，那么任意做一次哈希计算都可以达到这个目标，这就失去了“工作量证明”的意义。

当最快的、最接近难度目标值的哈希值产生后，该矿工可以向全网广播，如果同一时间获得超过 51%的节点认可，则意味着该矿工完成了一个新区块的创建，可以链接到区块链上。其他矿工再以这个区块为父区块，进行下一个区块的创建。

① 参考：http://news.cecb2b.com/2016/0424/3378735.shtml。

仅供非商业用途或交流学习使用

（2）这里所说的难度值（Difficulty）是指节点要运算出小于难度目标值的哈希值平均需要花费多长时间，也就是平均完成一次 PoW 计算的时间。在比特币网络中，目前设定为平均每 10 分钟会有节点成功计算出新的区块，不过这 10 分钟只是基于理论值的，实际上每个新区块产生的时间有可能只需要 6 秒（比如第 515 664 个区块的实际产生时间），也有可能需要 16 分钟（比如第 515 659 个区块的实际产生时间），但系统要保证平均 10 分钟的节奏不变。

TIPS:

在比特币系统中，难度值是动态调整的，目前每产生 2016 个区块会调整一次难度值。以每 10 分钟产生一个区块估算，大约每两周会调整一次难度值。

难度值的调整是在每个完整节点中独立自动发生的，具体机制可以参考 4.2.3 节，在此不再赘述。

目标值的大小与难度值成反比。比特币工作量证明的完成是以矿工计算出来的区块哈希值小于难度目标值且最接近难度目标值为标准的。

由于 PoW 具有一定的难度，因此无法预期哪个运算节点可以最快计算出新区块，这样就确保了交易验证的公平性。

5.1.7 随机数

在整个区块头的数据结构中，大部分是经过哈希运算后的固定值，唯一不是固定值的字段就是随机数（Nonce）。它是创建新区块的“钥匙”，其工作原理（以比特币为例）如下：当区块头及区块主体构建完成以后，挖矿（解数学题）就可以开始进行了，挖矿的目标就是通过不断改变区块头中的 Nonce 值再来哈希，直到对区块头使用 SHA256 算法得出的哈希值符合难度目标值的要求。

比特币挖矿的目标就是找到这样一个 Nonce 值，使得在这个值下的区块头的 SHA256 哈希值的输出必须小于难度目标值中设定的目标值。矿工通过不停地变更区块头中的 Nonce 值，并对每次变更后的区块头做双重 SHA256 运算（SHA256(SHA256(Block_Header))），将结果值与当前网络的目标值进行对比，如果小于目标值，则工作量证明完成，区块创建成功，该名矿工需要向全网络

仅供非商业用途或交流学习使用

广播这一包含 Nonce 值的区块，经过其他矿工验证通过后即获得全网络节点的认可，链入区块链成为新的区块，该名矿工会得到一定数量的比特币奖励，其他矿工则迅速以该区块为父区块，进行新的计算并试图“挖出”新的区块。

在挖矿过程中，由于每名矿工创建的新区块头中的时间戳都可能不一样，而且每名矿工选择进入本区块的交易集合不一样，区块头中的默克根也不一样，所以即使很多矿工都从 Nonce 值等于 0 开始累加寻找符合条件的哈希值，但是他们仍在各自不同的位置寻找，挖矿的过程是整个比特币网络所有矿工节点的计算能力加在一起寻找答案的过程，每名矿工都有找到答案的机会，但是拥有更高计算性能硬件或软件的矿工找到答案的概率更大。

在对区块头进行哈希运算的过程中，Nonce 值一般从 0 起步，每次加 1，然后不断进行哈希计算，直到完成目标。例如编号为 515671 的区块，其 Nonce 值为 224661451，区块的创建时间约为 9 分钟，说明这个区块在 9 分钟内完成了 Nonce 值递加和哈希过程累计 2.24 亿次以上。

5.2　区块链技术的算法

在区块链技术系统中，最重要的算法是哈希算法，它为区块链技术提供了数据效率和共识基础，并以超强的加密特性为区块链能够在匿名情况下产生信任、防止恶意篡改等性能提供了可靠的保证。

首先，共识机制的“挖矿”过程就是一个不断进行哈希运算的过程，用得到的哈希值去碰撞目标值，达到目标要求后碰撞成功，矿工获得创建新区块的机会。

其次，在区块对全网交易的封装过程中也采用了哈希算法，把大量的复杂交易信息进行一次又一次的哈希，分层进行两两合并，最终得到一个包含区块内全部交易信息的哈希值，实现了数据压缩和高效传播。

最后，在数字货币的支付过程中，支付方与接收方的钱包之间的通信所采用的公钥与私钥机制也多次用到了哈希算法，从两方面保证了系统的安全性，

仅供非商业用途或交流学习使用

即：公钥虽然公开发送但不能逆向解码；私钥由自己掌握，保证了传输数据的安全性。

图 5.6 说明了哈希算法在区块链技术中的基础地位。

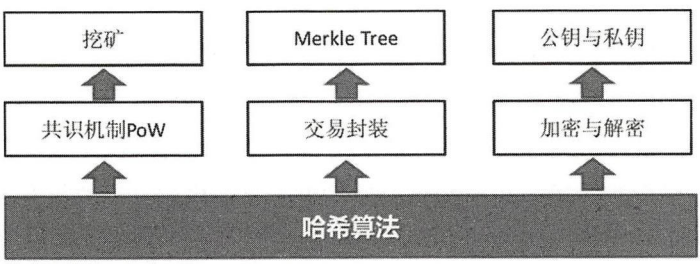


图 5.6　哈希算法在区块链技术中的基础地位

5.2.1　哈希算法概况

哈希算法，又称为哈希函数、散列函数，是指将任意长度的数据通过算法映射为较短的、固定长度的二进制值的过程。通过这一过程得到的较短的二进制值称为哈希值。

哈希算法最重要的特点是：它是一种单向的密码体制，也就是说，只能从数据开始，用哈希算法处理后得到哈希值，这一过程是不可逆的，没有办法从一个哈希值反向推算出原数据。所以，哈希算法相当于一个从明文到密文的不可逆加密过程。

TIPS:

哈希算法可以将任意长度的输入经过计算以后得到算法所设定的固定长度的输出，而且这个输出是唯一的，一个哈希值对应的只能是一个原文，不可能有同一个哈希值对应两个不同原文的情况。

更重要的是，只要原文发生一点小小的变动，其哈希计算的结果就会发生变动。正是由于哈希算法具有这些鲜明的特点，如单向不可逆、输出数据长度固定而且变短，以及对数据变动的敏感性，被大量用于数据完整性检验、数据加密等过程。常用的哈希算法包括 MD2、MD4、MD5、SHA-1、SHA-2 等。

在比特币为代表的区块链中，使用的是 SHA-2 系列中的 SHA256 算法。

仅供非商业用途或交流学习使用

SHA 的全称是 Secure Hash Algorithm，是一套逐步发展的加密哈希函数的总称，这些函数是由美国标准与技术局（National Institute of Standards and Technology）发布的，并作为美国联邦信息处理标准（FIPS）的组成部分。

这些函数分为如下几类。

（1）SHA-0：特指在 1993 年以“SHA”之名发布的 160 位哈希函数的原始版本，后因存在未公开的“显著缺陷”而被收回，并由经过修订的 SHA-1 取代。

（2）SHA-1：一种集成了早期 MD5 算法的 160 位哈希函数，由美国国家安全局（NSA）设计，并成为“数字签名算法”的组成部分。由于在其中发现了加密缺陷，所以在 2010 年后这一标准不再被大多数加密应用所接受。

（3）SHA-2：同样由美国国家安全局设计，包括两类相近的哈希函数——SHA256 和 SHA512。两者的主要区别在于哈希值的数据位数不同，SHA256 是 32 位的，而 SHA512 是 64 位的。根据位数不同，SHA-2 函数群包含 6 个哈希函数：SHA224、SHA256、SHA384、SHA512、SHA512/224 和 SHA512/256。

（4）SHA-3：这个哈希函数之前名为“Keccak”，是 2012 年经过公开竞赛（NSA 未参加）挑选出来的哈希函数，它同样支持 SHA-2 的数据位数，但其内部结构与其他 SHA 函数显著不同。

如表 5.3 所示为各类哈希函数的特点对比。

表 5.3　各类哈希函数的特点对比

<table>
<tr><th colspan="2">算　法</th><th>输出值位数（位）</th><th>内部数据位数（位）</th><th>数据块大小（位）</th><th>最大信息容量（位）</th><th>运算方式</th><th>安全性（位）</th></tr>
<tr><td colspan="2">MD5</td><td>128</td><td>128</td><td>512</td><td>无限制</td><td>与，异或，矩阵，32 位求模，或</td><td><64</td></tr>
<tr><td colspan="2">SHA-0</td><td>160</td><td>160</td><td>512</td><td rowspan="2">$2^{64}-1$</td><td rowspan="2">与，异或，矩阵，32 位求模，或</td><td><80</td></tr>
<tr><td colspan="2">SHA-1</td><td>160</td><td>160</td><td>512</td><td><80</td></tr>
<tr><td rowspan="6">SHA-2</td><td>SHA224</td><td>224</td><td rowspan="2">256</td><td rowspan="2">512</td><td rowspan="2">$2^{64}-1$</td><td rowspan="2">与，异或，矩阵，32 位求模，或</td><td>112</td></tr>
<tr><td>SHA256</td><td>256</td><td>128</td></tr>
<tr><td>SHA384</td><td>384</td><td rowspan="4">512</td><td rowspan="4">1024</td><td rowspan="4">$2^{128}-1$</td><td rowspan="4">与，异或，矩阵，64 位求模，或，移位</td><td>192</td></tr>
<tr><td>SHA512</td><td>512</td><td>256</td></tr>
<tr><td>SHA512/224</td><td>224</td><td>112</td></tr>
<tr><td>SHA512/256</td><td>256</td><td>128</td></tr>
</table>

续表

算　法		输出值位数（位）	内部数据位数（位）	数据块大小（位）	最大信息容量（位）	运算方式	安全性（位）
SHA-3	SHA3-224	224	1600	1152	无限制	与，异或，矩阵，非	112
	SHA3-256	256		1088			128
	SHA3-384	384		832			192
	SHA3-512	512		576			256
	SHAKE128	d（随机）		1344			min(d/2,128)
	SHAKE256	d（随机）		1088			min(d/2,128)

哈希算法的“安全性”是对其单向特性的一种测试结果的表示，即对某个确定的哈希值，通过一次次地列举不同的输入值，来测试能否找到和原输入值相同的内容，从而使哈希算法单向特性（不同的输入值对应不同的哈希值）不成立。如果穷举次数达到某一数量级后碰撞成功，则这个数量级就是该哈希算法的安全性指标，意即当超过这个指标时可能通过暴力穷举反向找到输入值。

TIPS:

在上述表格中，以 SHA-1 为例，其最大信息容量为 $2^{64}-1$ 位，表示对于最多不超过 2^{64} 位的信息量，使用 SHA-1 算法可以产生一个 160 位的摘要，并且安全性可以达到在随机碰撞不超过 2^{80} 次时，不会找到同样的散列值。这是一个巨大的数字，在当前情况下可能性为零。

5.2.2 SHA256 算法

2002 年 8 月，美国标准与技术局正式发布了新的安全哈希算法，也就是 SHA-2 系列，其中就包括 SHA256 算法。这种算法被广泛应用于比特币及其他区块链应用中，用于挖矿、公钥和私钥加密等过程。

在美国标准与技术局公布的FIPS 180-2标准中，共定义了4种安全哈希算法：SHA1、SHA256、SHA384 和 SHA512。这 4 种算法都是迭代的、单向的哈希函数，可以用于处理信息并产生精简的“信息摘要”。这些算法可以用来验证一条信息的完整性，因为对信息的任何改变（不分大小，哪怕是变动一个标点符号）都将产生不同的信息摘要。基于这一点，在数据签名、信息认证的产生及验证等方面，安全哈希算法都具有广阔的应用前景。下面重点介绍 SHA256 算法。

1. 基础定义

SHA256 算法是计算机算法的一种，它的处理对象也是最基础的计算机二进制字符，因此，在 SHA256 算法中也适用关于二进制位的基础定义，如表 5.4 所示。

表 5.4　SHA256 算法的基础定义

名　　称	含义或用法
位（bit）	一个二进制数位，值为 0 或 1
字节（Byte）	8 位（bit）组成 1 字节
字（word）	根据安全哈希算法，由 32 位（4 字节）或 64 位（8 字节）组成

2. 参数

如表 5.5 所示的参数用于算法定义。

表 5.5　SHA256 算法的参数

参　　数	含义或用法
$a,b,c,\cdots,h$	在 w 位的字用于计算哈希值 $H^{(i)}$时用到的工作变量
$H^{(i)}$	第 i 位哈希值，$H^{(0)}$是第一位哈希值，$H^{(N)}$是最后一位哈希值，决定信息摘要
$H_j^{(i)}$	第 i 位哈希值的第 j 位字，如$H_0^{(i)}$表示哈希值 i 的最左边的“字”
K_t	用于哈希计算的 t 循环的常量
K	在填补阶段添加到信息中的 0 的个数
l	信息 M 的长度，单位为位
m	信息块 $M^{(i)}$的数据位数
M	用于哈希的信息
$M_j^{(i)}$	第 i 个信息块的第 j 位字，如$M_0^{(i)}$表示信息块 i 最左边的那个“字”
n	一个“字”在计算过程中旋转或移位的位数
N	填补后的信息中的信息块数
T	在哈希计算中临时 w 位的“字”
w	一个“字”的位数
W_t	在信息表的第 t 个 w 位的“字”

3. 符号

如表 5.6 所示的符号用于安全哈希计算定义，运算对象都是 w 位的“字”。

表 5.6　SHA256 计算符号

符　　号	含义或用法
∧	按位与运算
∨	按位或（逻辑或）运算
⊕	按位异或（互斥或）运算
¬	按位补码运算
+	模 2^w 加法
≪	左移位运算，$x \ll n$ 表示去掉“字”x 最左边的 n 位，然后在右边用 n 个 0 补充
≫	右移位运算，$x \gg n$ 表示去掉“字”x 最右边的 n 位，然后在左边用 n 个 0 补充

4. 记数法

一个十六进制数字是指集合{0,1,…,9,a,…,f}中的一个元素。一个十六进制数字代表一个 4 位的字符串，例如，十六进制“7”代表 4 位字符串“0111”，十六进制“a”代表 4 位字符串“1010”。

一个“字”是一个 w 位的字符串，可以表示为一个十六进制数字的序列。要把“字”转换为十六进制数字，每 4 位字符串要转换为它对等的十六进制数字。

如 32 位字符串“1010 0001 0000 0011 1111 1110 0010 0011”可以表示为十六进制数字“a103fe23”；64 位字符串“1010 0001 0000 0011 1111 1110 0010 0011 0011 0010 1110 1111 0011 0000 0001 1010”可以表示为十六进制数字“a103fe2332ef301a”。

5. 字运算

在 SHA256 算法中定义了以下“字”运算如表 5.7 所示。

表 5.7　SHA256 算法中的“字”运算

字运算及符号（表达式）	含义或用法
按位逻辑运算：∧，∨，⊕	按位进行逻辑运算
模 2^w 加法	对于正整数 U 和 V 来说，“U 模 V 计算”就是取 U 除以 V 的余数。 运算 $x+y$ 如下定义：“字”x 和 y 代表整数 X 和 Y，其中 $0 \leqslant X<2^w$，$0 \leqslant Y<2^w$。 计算 $Z=(X+Y) \bmod 2^w$，$0 \leqslant Z<2^w$，Z 转换为一个“字”z，令 $z=x+y$
右移位运算 $SHR^n(x)$	定义如下（其中 x 是一个 w 位的“字”，$0 \leqslant n<2^w$）：$SHR^n(x)=x \gg n$

续表

字运算及符号（表达式）	含义或用法
“循环右移”运算 $\mathrm{ROTR}^n(x)$	x 是一个 w 位的“字”，n 是一个整数，$0 \leqslant n < w$， $\mathrm{ROTR}^n(x)=(x \gg n) \vee (x \ll w-n)$ $\mathrm{ROTR}^n(x)$等价于把 x 的 n 位循环移动到右边
“循环左移”运算 $\mathrm{ROTL}^n(x)$	x 是一个 w 位的“字”，n 是一个整数，$0 \leqslant n < w$， $\mathrm{ROTR}^n(x)=(x \ll n) \vee (x \gg w-n)$ $\mathrm{ROTL}^n(x)$等价于把 x 的 n 位循环移动到左边

在上述运算中，存在以下等价关系（其中 w 是固定的）：

$$\mathrm{ROTL}^n(x) \approx \mathrm{ROTR}^{w-n}(x)$$

$$\mathrm{ROTL}^n(x) \approx \mathrm{ROTR}^{w-n}(x)$$

6. 函数

SHA256 算法使用 6 个函数，每个函数运行在 32 位“字”（用 x、y 和 z 表示）的基础上，结果产生一个新的 32 位“字”。

$$\mathrm{Ch}(x,y,z) = (x \wedge y) \oplus (x \wedge z)$$

$$\mathrm{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\{256\}}(x) = \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x)$$

$$\sum_1^{\{256\}}(x) = \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x)$$

$$\sigma_0^{\{256\}} = \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x)$$

$$\sigma_1^{\{256\}} = \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x)$$

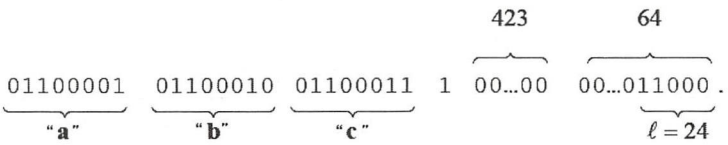
7. 预处理过程

预处理发生在哈希计算开始前，由 3 步组成：填补信息 M；把信息 M 分割成信息块；设定初始哈希值$H^{(0)}$。

（1）填补信息的过程是这样的：假定信息 M 的长度为 l 位，在末尾加上“1”，以及 k 个“0”，其中 k 是等式 $l+1+k \equiv 448 \bmod 512$ 的最小非负解。

（2）添加一个 64 位的块，等于把 l 用二进制表示的值。例如，（8 位 ASCII

码）信息“abc”的长度为 8×3=24，所以这个信息先加上 1 位“1”，再加上 448−(24+1)=423 个 0，最终填补后信息的长度变成 512 位，如下：



（3）分割信息块的过程是将填补后的信息分割成 N 个 512 位的块，分别标识为$M^{(1)},M^{(2)},\cdots,M^{(N)}$。由于输入块的 512 位可以表示为 16 个 32 位的“字”，则信息块 i 的第 1 个 32 位“字”表示为$M_0^{(i)}$，下一个 32 位“字”是$M_1^{(i)}$，一直到$M_{15}^{(i)}$。

（4）设定初始哈希值（$H^{(0)}$）。

每种安全哈希算法在开始哈希计算前，必须设定初始哈希值（$H^{(0)}$）。$H^{(0)}$的位数和个数取决于最终的信息摘要的要求。SHA256 算法的初始哈希值由 8 个 32 位“字”组成，用十六进制表示如下：

$$H_0^{(0)} = \text{6a09e667}$$

$$H_1^{(0)} = \text{bb67ae85}$$

$$H_2^{(0)} = \text{3c6ef372}$$

$$H_3^{(0)} = \text{a54ff53a}$$

$$H_4^{(0)} = \text{510e527f}$$

$$H_5^{(0)} = \text{9b05688c}$$

$$H_6^{(0)} = \text{1f83d9ab}$$

$$H_7^{(0)} = \text{5be0cd19}$$

8. SHA256 的安全哈希计算

SHA256 算法用于对长度为 l 的信息 M 进行哈希计算，其中 $0 \leqslant l < 2^{64}$。在哈希计算的过程中包括：①有 64 个 32 位“字”的信息表；②8 个工作变量，每个都为 32 位；③哈希值为 8 个 32 位“字”。SHA256 算法的最终结果是一个 256 位的信息摘要。



信息表的“字”标记为$W_0,W_1,\cdots,W_{63}$，8 个工作变量标记为 a,b,c,d,e,f,g,h。哈希值的“字”标记为$H_0^{(i)},H_1^{(i)},\cdots,H_7^{(i)}$，每个保存初始哈希值$H^{(0)}$，在处理每个信息块之后，由后续的过程哈希值取代，最终的哈希值表示为$H^{(N)}$。SHA256 计算过程使用了两个临时的字 T_1 和 T_2。

当预处理完成后，每个信息块$M^{(1)},M^{(2)},\cdots,\mathrm{M}^{(N)}$按以下步骤依次处理。

```
For i=1 to N
{
1. 准备信息表{W_t}
```

$$W_t=\begin{cases}M_t^{(i)} & 0\leqslant t\leqslant 15\\ \sigma_1^{\{256\}}(W_{t-2})+W_{t-7}+\sigma_0^{\{256\}}(W_{t-15})+W_{t-16} & 16\leqslant t\leqslant 63\end{cases}$$

```
2. 用第（i-1）个哈希值来初始化 8 个工作变量：a,b,c,d,e,f,g,h
```

$$\begin{aligned}a&=H_0^{(i-1)}\\ b&=H_1^{(i-1)}\\ c&=H_2^{(i-1)}\\ d&=H_3^{(i-1)}\\ e&=H_4^{(i-1)}\\ f&=H_5^{(i-1)}\\ g&=H_6^{(i-1)}\\ h&=H_7^{(i-1)}\end{aligned}$$

```
3. For t=0 to 63
{
```

$$T_1=h+\sum\nolimits_1^{\{256\}}(e)+Ch(e,f,g)+K_t^{\{256\}}+W_t$$

$$T_2=\sum\nolimits_0^{\{256\}}(a)+Maj(a,b,c)$$

$$\begin{aligned}h&=g\\ g&=f\\ f&=e\\ e&=d+T_1\\ d&=c\\ c&=b\\ b&=a\\ a&=T_1+T_2\end{aligned}$$

```
}
4. 计算第 i 个过程的哈希值H^(i)
```

$$\begin{aligned}H_0^{(i)}&=a+H_0^{(i-1)}\\ H_1^{(i)}&=b+H_1^{(i-1)}\\ H_2^{(i)}&=c+H_2^{(i-1)}\\ H_3^{(i)}&=d+H_3^{(i-1)}\end{aligned}$$



$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

}

重复 1～4 的过程 N 次（处理完信息$M^{(N)}$）之后，得到信息 M 的 256 位信息摘要如下：

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

综合来看，SHA256 算法（以及另外 3 种安全哈希算法）先对原始信息进行填补和分块，然后进行数据计算，使原始信息的细微变动也能够在信息摘要中反映出来；在数据计算过程中，通过不断地进行逻辑运算与移位运算，确保结果不可逆推及其唯一性，使数据的加密特性得到保障。当具备这两方面的特征后，SHA256 等安全哈希算法成为区块链技术所选择的最适用的加密算法。

当然，上述计算过程对于普通人来说比较复杂和难以理解，目前有一些网站提供了集成的将任意的普通信息转换为 SHA256 哈希值的功能，如 http://www.xorbin.com/tools/sha256-hash-calculator，在信息框中输入任意一段信息，单击“计算 SHA256 哈希值”按钮，即可生成一段哈希值，也就是最终的信息摘要哈希值。例如，我们输入一段英语新闻：

While the blockchain technology used to power new cryptos may not be as sexy as the digital tokens that seem to triple in value in the blink of an eye, blockchain is serving as the backbone to explore dynamic new business models far beyond finance, from healthcare and food safety to government and global trade.

The beauty of blockchain rests on its ability to create a single, shared version of a trusted ecosystem. A distributed ledger, blockchain networks enable the verification and storage of transactional records to be divvied up among a network of computers. There is no central authority or middleman that transactions must pass through. Each node on the network has an exact copy of the record, making it near impossible for anyone to tamper with or change it.

经哈希计算完成后得到的 SHA256 哈希值为 0ad699f0ff4c1c607284deee76cbd13a1ab506081b16b24d021113da2e61c4c4（其中每个都为十六进制的数字，



每个数字表示 4 位的二进制码，共有 64 个数字，对应表示最终的 64×4=256 位信息摘要）。

通过对比可以看出，哈希计算在信息摘要方面的功能相当有效和强大。

5.3　椭圆曲线加密算法

在介绍比特币交易流程时，我们曾经提到一项加密技术——SECP256K1，它是在由私钥产生公钥过程中的另一种基本算法，在比特币网络的运行过程中，在交易数据的加密与解密方面也起着非常基础的作用，与 SHA256 算法一起构成了比特币这一加密数字货币体系的基础算法。

SECP256K1 算法是椭圆曲线加密算法的一种，中本聪在开发比特币系统时选择了这种算法。这种算法的参数固定，避免了采用随机种子法容易被安插后门的风险，并且用 SECP256K1 所计算出的可用的比特币地址超过 2^{160} 个，相比之下，地球上的沙子全部数量不超过 2^{70} 粒，银河系内的星星数量不超过 2^{35} 颗，所以 SECP256K1 算法所产生的比特币地址数量极大，难以被采用碰撞的方法破解，在保证计算速度的前提下提供了足够的安全性。

TIPS:

在 SECP256K1 的名称中，SEC 是“Standards for Efficient Cryptography”的缩写，P 表示其定义域是有限域（质数域）F_p，256 表示定义域的位数，K 表示参数与 Koblitz 曲线有关（另一种采用随机种子，用 R 表示），1 表示序数。

5.3.1　椭圆曲线加密算法的特点

椭圆曲线加密算法的英文全称是 Elliptic Curve Cryptography，英文缩写为 ECC，它是在有限域内、在椭圆曲线的代数结构的基础上来对公钥进行加密的一种算法，这是比特币中采用的一种加密算法，目的在于确保只能由正确的拥有者来支付资金。



公钥的加密过程一般都是通过解出难度较高的数学问题来实现的。早期的公钥系统常用加密方法是把一个很大的整数分解成两个或多个大素数；而椭圆曲线加密算法采用的是“点乘”运算，即沿着椭圆曲线对一个点进行连续性相加。与其他加密算法相比，椭圆曲线加密算法能够用更小的密钥实现相同的安全性，降低了数据存储和传输的需求，比如，一个 128 位的椭圆曲线加密密钥能够提供的安全性与一个 3072 位的 RSA 密钥能够提供的安全性一样。

椭圆曲线加密算法最早由美国华盛顿大学的数学教授 Neal Koblitz 和普林斯顿大学的教授 Victor S. Miller 在 1985 年开始独立使用，2004—2005 年间得以广泛应用。基于其特点，这种算法常被用于加密、数字签名及伪随机码生成器等方面。

基于有限质数域F_p的椭圆曲线加密算法，根据其椭圆曲线所选参数的不同，共有 8 类，如表 5.8 所示。

表 5.8　椭圆曲线的不同类型

参　数　集	密码强度（位）	定义域位数（位）	相当于 RSA 或 DSA 加密算法强度（位）	Koblitz 曲线（K）或随机种子（R）
SECP192K1	96	192	1536	K
SECP192R1	96	192	1536	R
SECP224K1	112	224	2048	K
SECP224R1	112	224	2048	R
SECP256K1	128	256	3072	K
SECP256R1	128	256	3072	R
SECP384R1	192	384	7680	R
SECP512R1	256	512	15360	R

其中，SECP256K1 就是比特币系统选用的椭圆曲线加密算法的参数集。

5.3.2　椭圆曲线加密算法的数学原理

用于加密的椭圆曲线[①]是在有限域上的一条平面曲线，这条平面曲线由满

① 参考：https://www.certicom.com/index.php/10-introduction。



足一个椭圆曲线方程的点及一个无限远的点（用∞表示）组成。有限域是指对于一个集合 F，其中的元素在经过加法和乘法计算后，得到的结果仍然包含在集合 F 的元素范围中。在椭圆曲线加密算法中使用了两种类型的有限域：一种是质数域F_p，p 为一个质数；另一种是基于特征值 2 的有限域F_2m，即关于2^m的有限域，其中 m>1，F_2m又称为二元扩域。

在实数域和质数域内，椭圆曲线的方程表达式为：$y^2 = x^3 + ax + b$。

在方程中，x,y,a,b 都为实数，不同的赋值可以画出不同的椭圆曲线。比如，当 a=−4,b=0.67 时，其椭圆曲线如图 5.7 所示。

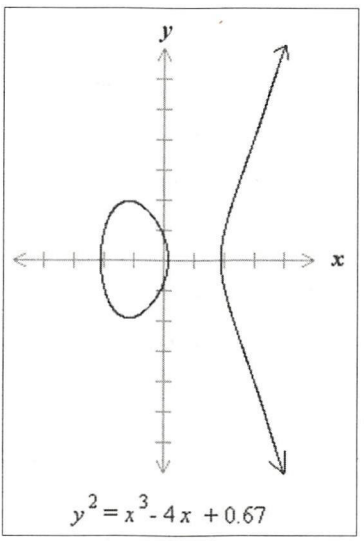


图 5.7　椭圆曲线示意图

满足$4a^3 + 27b^2 \neq 0$的点组成了椭圆曲线群，每个椭圆曲线群都是加法群，因为在椭圆曲线中定义了一种几何的加法运算：P+Q=R，这种加法与我们常规所认识的加法并不相同。另外，在椭圆曲线群中对点的映射进行了定义，即对于点 $P(x_P,y_P)$，它关于 x 轴的映射为点$-P(x_P,-y_P)$。

在此基础上，P+Q=R 的详细过程是这样的：假定在椭圆曲线上有两个不重合的点 P,Q，且 $P \neq -Q$，连接 P,Q 作一条直线，这条直线交椭圆曲线于一点$-R$，这个$-R$ 点是 R 点关于 x 轴的映射，规定 P+Q=R，这就是椭圆曲线中采用几何定义的“加法”，如图 5.8 所示为一个例子。



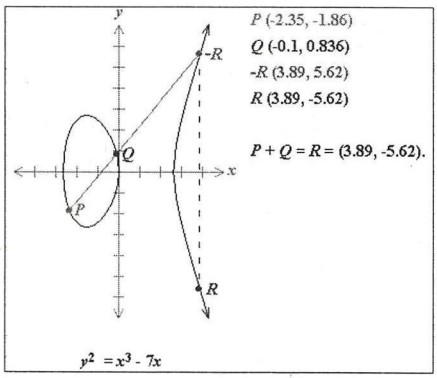


图 5.8　椭圆曲线加法示意图

但对于 P 点和$-P$ 点，连接这两个点的直线与椭圆曲线没有第三个交点，因此不能像上述的规则那样把 P 和$-P$ 相加。为此，在椭圆曲线群中包括一个无限点 O，并定义 $P+(-P)=O$，这个等式经过变换后的结果就是 $P+O=P$，定义域为椭圆曲线群。O 点被称为该椭圆曲线群的“加法恒等元”，所有的椭圆曲线群都有一个加法恒等元。

如图 5.9 所示为在一个椭圆曲线群中 $P+(-P)$的图示。

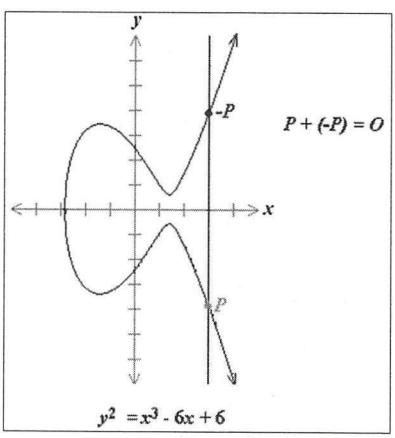


图 5.9　椭圆曲线群示意图

对一个点 P 倍加，即 $P+P$，在椭圆曲线上的定义是：先从 P 点作椭圆曲线的切线，如果 P 点的 y 坐标不为 0，则该切线会交椭圆曲线于另一个点$-R$，再作点$-R$ 关于 x 轴的映射点 R，定义 $P+P=2P=R$。如图 5.10 所示为对一条椭圆曲线上的点 P 的倍加过程。



仅供非商业用途或交流学习使用

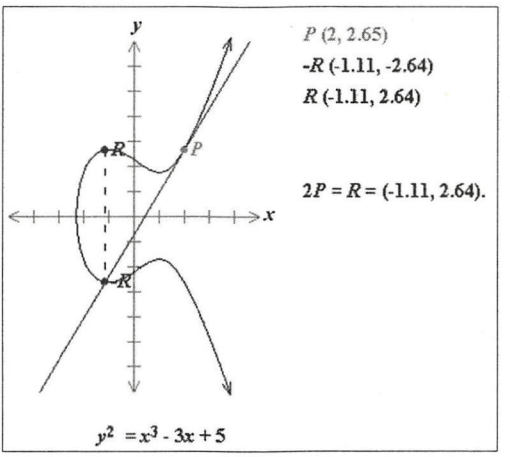


图 5.10　椭圆曲线倍加示意图

那么，如果 P 点在 y 轴上的坐标值为 0，那么对这样的点该怎样倍加呢？对于这样的 P 点，从 P 点作椭圆曲线的切线，这条切线将总是垂直于 x 轴，与椭圆曲线没有交点，因此，定义这条切线在无限远处与椭圆曲线相交于点 O，即当 y_P=0 时，定义 $2P=O$，交点在无限远处。

如图 5.11 所示为 y_P=0 时的倍加图例。

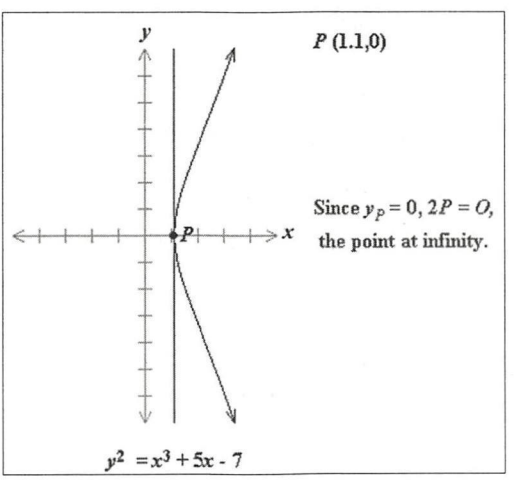


图 5.11　y_P=0 时的倍加图例

在 y_P=0 的情况下，因为 $2P=O$，所以 $2P+P=P+O=P$，即 $3P=P$；同理可以推导出 $4P=O$，$5P=P$，$6P=O$，$7P=P$，以此类推。

仅供非商业用途或交流学习使用

加密算法的一大特性是算法群的数据点是有限的。基于实数的加密算法，由于存在舍入错误，导致速度慢而且不准确。因此，基于有限域F_p和F_2m 的椭圆曲线群在实践中得到了应用和发展。

TIPS:

域F_p表示数据取值范围为 0～p-1，算到除以 p 的余数时结束。比如F_{23}域由 0～22 的整数组成，并且这个域中的任意运算结果的范围也是 0～22 之间的整数。

位于基础域F_p内的椭圆曲线由位于域F_p内的不同变量 a 和 b 组成，椭圆曲线包含所有的点 $P(x,y)$，其中 x,y 是域F_p内的取值，这些点也都满足椭圆曲线方程模 p 的计算。

例如，$y^2 \mod P=x^3+ax+b \mod p$，如果 a,b 都位于域F_p内，则椭圆曲线方程的基础域也在域F_p内。

此时，只要x^3+ax+b 不含重复因式（或者等价于 $4a^3+27b^2 \mod p$ 不等于 0），这条椭圆曲线就可以组成一个群，在这个群中既包含椭圆曲线上所有的点，也包含一个特殊的点，即无限远的 O 点，也就是加法恒等元。

下面举一个例子进行详细说明。比如一条椭圆曲线的基础域为F_{23}，当 a=1,b=0 时，椭圆曲线的方程为$y^2 = x^3 + x$，点(9,5)满足方程的要求，因为

$$y^2 \mod p = x^3 + x \mod p$$

$$25 \mod 23=729+9 \mod 23$$

$$25 \mod 23=738 \mod 23$$

$$2=2$$

按此方法，可以求出所有满足椭圆曲线方程的 23 个点，即：(0,0)，(1,5)，(1,18)，(9,5)，(9,18)，(11,10)，(11,13)，(13,5)，(13,18)，(15,3)，(15,20)，(16,8)，(16,15)，(17,10)，(17,13)，(18,10)，(18,13)，(19,1)，(19,22)，(20,4)，(20,19)，(21,6)，(21,17)。

在二维坐标轴上描出这 23 个点，如图 5.12 所示。



仅供非商业用途或交流学习使用

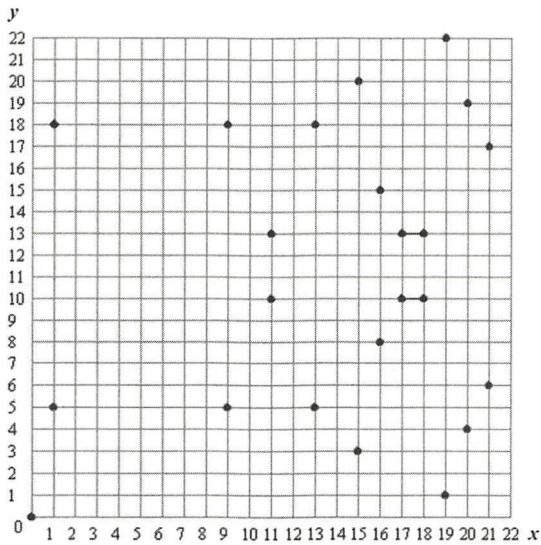


图 5.12　椭圆曲线解集的二维分布图

从图 5.12 中可以看出，每个 x 值都有两个点，而且尽管看起来是随机分布的，但实际上这些点是关于 y=11.5 对称的。另外，根据实数的椭圆曲线的特性，每个点都有一个负的、关于 x 轴的映射点。

有限域F_p内的椭圆曲线与实数域内的椭圆曲线有几个主要区别。有限域F_p内的椭圆曲线群拥有数量有限的点，这一特点正是加密算法所需要的，这些曲线都是由离散的点组成的，怎样把这些点连成一条曲线，或者这些曲线之间应用了怎样的几何关系，这些都不清楚。因此，应用于实数域内的椭圆曲线群的几何规则并不能应用到有限域F_p内的椭圆曲线群，不过二者的代数运算规则是相同的。另外，与实数域内的椭圆曲线不同的是，有限域F_p内的计算不会出现舍入错误，这正是加密算法所需要的。

有限域F_p内的椭圆曲线的代数运算规则是这样的。

（1）两个不同的点 P、Q 相加。

对于点 $P(x_P,y_P)$，其对应的负值为$-P(x_P,y_P \bmod P)$。如果 P 和 Q 为不同的点且 P 不等于$-Q$，则 $P+Q=R$，R 点的坐标值为(x_R,y_R)，此时连接 P 点和 Q 点的直线的斜率 s 为

$$s=(y_P-y_Q)/(x_P-x_Q) \bmod p$$

仅供非商业用途或交流学习使用

则 $x_R=s^2-x_P-x_Q \mod p$，$y_R=-y_P+s(x_P-x_R) \mod p$

计算出了 x_R,y_R，就可以确定 R 点所在的位置了。

（2）P 点倍加。

当 y_P 不为 0 时，P 点的倍加 $2P=R$，则有：

$$s=(3x^2+a)/(2y_P) \mod p$$

$$x_R=s^2-2x_P \mod p，\ y_R=-y_P+s(x_P-x_R) \mod p$$

基础域在二元扩域F_2m内的椭圆曲线，其代数算法与上述算法的原理相似，也包括异点加法和同点倍加两种运算，只是椭圆曲线方程的表达式不一样，是一个二元的表达式：$y^2+xy=x^3+ax+b$。由于比特币系统只采用了质数域F_p内的椭圆曲线加密算法，因此本章不再对二元扩域F_2m内的椭圆曲线加密算法进行详细说明。

5.3.3 椭圆曲线加密算法的加密原理

应用于比特币系统中的椭圆曲线加密算法[①]，在上述数学原理的基础上，充分应用了椭圆曲线加密算法的单向特性，由私钥可以产生公钥，而反过来要想由公钥推算出私钥则不可能；用公钥加密的信息可以用私钥来解密；而用私钥签名的信息则由公钥来验证，验证通过后说明该信息确实是由私钥持有人所发布的。

在椭圆曲线加密算法中，对于椭圆曲线$y^2=x^3+ax+b$，规定 a,b 都取自质数域F_p，且满足 $4a^3+27b^2\neq 0$（mod p），这条曲线上的点 $P=(x,y)$的集合组成了基于有限域（质数域）F_p的椭圆曲线域。公钥就是该曲线上的某个点 Q 的二进制表示，对于给定的点 P，再选取曲线上的任意一点 k，计算出 $Q=kP$，则可以由 k、P 计算出 Q，而在椭圆曲线的计算规则中无法由 Q、P 计算出 k（私钥）来。

为了描述特定的椭圆曲线域，需明确 6 个参数：$T=(p, a, b, G, n, h)$。

① 参考：http://www.8btc.com/btc_ecc_dsa_a。

仅供非商业用途或交流学习使用

p：代表有限域 F_p 的那个质数。

a,b：椭圆曲线方程的参数。

G：在椭圆曲线上任意选取的一个基点 $G=(x_G, y_G)$。

n：G 在 F_p 中规定的序号，它是一个质数。

h：余因式，控制选取点的密度，h =椭圆曲线上点的总数/ n。

在比特币系统选用的 SECP256K1 算法中，各参数的具体值如下：

```
p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
FFFFFC2F
= 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
a = 0,  b = 7
G =04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B
16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F
FB10D4B8
(当 G 点用压缩形式表示时，其值为 02 79BE667E F9DCBBAC 55A06295 CE870B07
029BFCDB 2DCE28D9 59F2815B 16F81798)
n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C
D0364141 (在比特币系统中规定，私钥的取值范围不能超过 n)
h = 01
```

具体的加密和解密过程如下：

（1）用户 A 先选出一条椭圆曲线 $E(a,b)$，然后在这条椭圆曲线上选取任意一点 G，称为基点（Base Point）。

（2）在椭圆曲线上选取任意一点 k，k 为小于 n 的整数，同时把 k 作为用户 A 的私钥，且根据椭圆曲线的代数算法，可以找到一点 Q，$Q=kG$，Q 即公钥。

（3）用户 A 将 $E_p(a,b)$和点 K、G 传给用户 B。

（4）用户 B 在收到用户 A 发来的公钥等信息后，将待传输的明文编码到 $E_p(a,b)$上的一点 M，并产生一个随机整数 r。

（5）用户 B 计算点 $C_1=M+rK$,$C_2=rG$。

（6）用户 B 将 C_1、C_2 传给用户 A。

（7）用户 A 在接收到信息后，计算 C_1-kC_2。

```
C1-kC2=M+rK-k(rG)=M+rK-r(kG)=M
```



仅供非商业用途或交流学习使用

再对点 M 进行解码就可以得到信息原文，完成一次加密通信过程。

其保密性体现在：如果有一个偷窥者 H 截取了用户 A 和 B 之间的通信信息，那么他只能截获到 $E_p(a,b)$、K、G、C_1、C_2。因为根据椭圆曲线加密算法的计算规则，通过 K、G 求 k 或通过 C_2、G 求 r 都是无法完成的，因此，H 无法得到 A、B 间传送的信息原文。椭圆曲线加密示意图如图 5.13 所示。

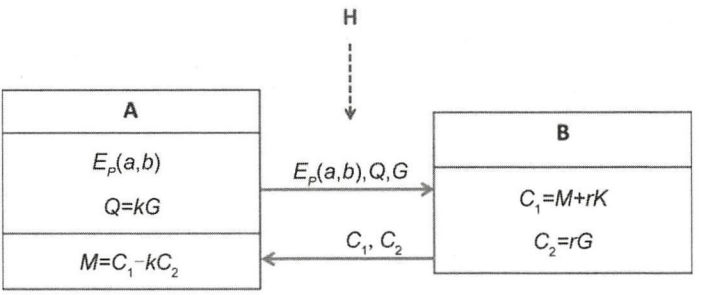


图 5.13　椭圆曲线加密示意图

5.4　区块链的运行机制

从比特币系统分析其底层的区块链技术，可以看到区块链技术的设计理念：区块链系统对交易信息进行连续不断的固化和透明化，形成全网可以掌握的分布式账本（或数据），由此产生陌生人之间的信任，而需要投入算力资源的共识机制确保整个系统的游戏规则是“一分耕耘，一分收获”，实现了在硬件层面的单一公平，最终也促进了网络中的交易者与网络的维护者之间形成共生与信任的良好关系，解决了传统交易过程中必须借助中心化机构或中介机构来产生信任的不足，有利于提升社会的总体交易效率。

TIPS:

在区块链网络中存在着海量的数据传输，永不停息地运行着消耗巨大算力的哈希计算，虽然看上去复杂多变，但实际上其内在逻辑只有两种类型：①区块链用户的交易与交流；②区块链维护者的“造链”过程。

仅供非商业用途或交流学习使用

以比特币系统为例，其区块链的运行流程可以分为两类：一是比特币用户通过比特币钱包进行了交易，匿名进行比特币的支付转移，或者在比特币以外的联盟链、私有链等侧链上，对信息进行压缩后（实际上就是用哈希算法对信息进行处理，取得信息摘要），发送到比特币的交易网络中来，利用比特币的全球公共区块链特性，实现信息的最终固化和不可篡改、安全性和可靠性；二是矿工进行算力竞争，不断进行区块链系统内的交易信息收集和验证，封装交易信息，在“争抢”造币权的过程中达成共识、实现造币，并顺序链接起来，使比特币系统可以不断运转下去。

如同中国的太极图，“太极生两仪，两仪生四象，四象生八卦”，在元规则建立后，随着元规则的运转和流动，产生了整个系统化体系。区块链系统的总体世界庞大而繁杂，但其底层的流程就是这两类，经过这两类流程不断地运行，将人类社会的交换、激励、维护、验证等功能分散到网络和机器的数据、信息和元规则上来，构建了一个具有颠覆意义的信用传递系统。

5.5　区块链的交易机制

当前最成熟的区块链应用是比特币系统，其次是其他各类虚拟币，再次是正在发展的基于以太坊而开发的与适用于不同应用场景的智能合约等各类产品，所有这些应用区块链技术为大众提供服务的产品都立足于运用区块链技术的优势来促进交易效率的提升：比特币及其他各种数字货币是为了替代现实世界中的货币及其清算系统，提高交易时效，降低交易成本；智能合约则实现了当条件成立时自动触发交易的完成……所以，交易流程是区块链技术运作流程中最主要的流程。

5.5.1　交易流程

交易流程的组成要素包括：①卖方和买方；②价值的交换或传递；③交易标的。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

在区块链系统中，通过算法设计和数据处理方式，使复杂的交易在进入区块链的过程中不断分解，直至成为最基本的若干个基础交易，只有具备上述 3 个方面的组成要素才能由区块链技术将其转化为进入区块的哈希散列值，链入分布式的透明“账本”中。

区块链中的交易可以发生在全部用户之间，既包括普通的用户，也包括系统的维护者“矿工”；同时卖方、买方的定位也不是固定的，根据交易的不同是在不断切换的；另外，卖方和买方也可以有多个，分别位于交易的两侧。所以，对于区块链中的卖方和买方而言，首先需要交易双方在区块链系统中进行身份的确认，也就是说拥有一个在区块链系统中唯一的且能够最终被验证和访问到的身份。

以比特币为例，目前在比特币系统中进行交易、甚至开展挖矿的用户只需拥有一个虚拟的身份，这个身份实际上是一个虚拟地址，其特点是使用方便、匿名，同时可以接收和支付比特币，因此需要具有较高的安全性。但普通用户并不需要、也没有能力直接与底层代码打交道，为此，比特币系统及很多第三方区块链服务商都给用户提供了一款软件，形象地称之为“钱包”。如同现实生活中的钱包一样，用户可以在“钱包”软件中存放自己拥有的比特币及私钥。

通过“钱包”软件，每个用户可以创建大量的虚拟地址，每个地址对应着一些比特币，并且每个地址都有对应的私钥，有了私钥才可以对相应的比特币进行支付或转账等操作。比特币地址是一段由数学算法生成的 27～34 位长度的字符串，一般以数字“1”或者“3”开头。每个比特币地址都对应着一个比特币私钥。比特币私钥也由一串字符组成，一般以数字“5”开头。比特币钱包的具体结构如图 5.14 所示。

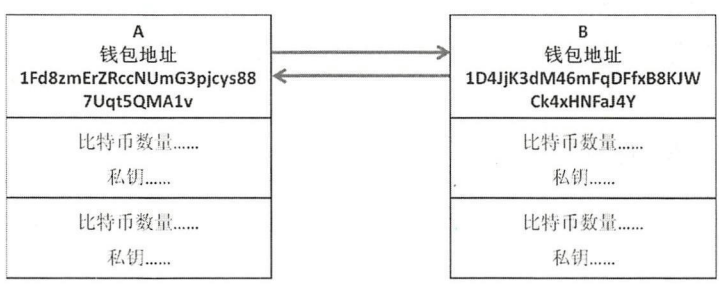


图 5.14　比特币钱包的具体结构

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

TIPS:

比特币地址和私钥总是成对出现的，比特币私钥是唯一证明用户对该私钥所对应地址拥有控制权的依据。一旦用户忘记、丢失或让他人窃取了私钥，那么该用户将失去对这一地址上的比特币的控制权，相当于丢失了这些比特币。

此外，由比特币的私钥可以生成该私钥对应的比特币地址，进而可以使用该地址上的比特币，但是由比特币地址不能计算出该地址所对应的私钥。因此，比特币用户一定要收藏和备份好比特币钱包，保护好私钥。

5.5.2　比特币钱包

根据介质形式，比特币钱包分为 3 类：比特币客户端钱包、比特币网络 Web 钱包和比特币手机钱包。

比特币客户端钱包是一款安装到个人电脑上的软件，常见的客户端钱包有官方钱包 bitcoin-core、MutiBit 等，这些钱包软件都有针对不同电脑操作系统如 Windows、Linux、Mac 的不同版本。官方钱包 bitcoin-core 的安全性、稳定性较高，但启动速度慢，第一次启动时需要同步完整的比特币数据块，数据块的容量达到 145GB，受到网络带宽的影响需要 1～3 天才能同步完成。不过，只需同步一次，以后再使用时不需要重新全部同步，仅同步新增内容即可。

因为比特币客户端钱包启动速度慢，需要同步大量数据，所以很多第三方企业提供了比特币网络 Web 钱包，应用比较广泛的网络钱包有 Blockchain、Coinbase 等。由于不是直接对应于比特币网络的，而是通过第三方的通道访问和使用用户的钱包的，因此，一旦这种钱包服务商出现了问题，很可能导致用户出现大量的比特币损失。另外，bitcoin-core 也推出了轻量级钱包，允许用户在具备安全可靠的连接方式以连接到全网节点时，以快速、简单的方式使用自己的比特币钱包。

比特币手机钱包是随着移动互联网的快速发展而出现的，主要应用于手机上，包括苹果 iOS 系统和 Android 系统两种版本，比如苹果推出的 iOS 10 系统就支持用户使用 Circle 的比特币支付系统。

目前常见的比特币钱包软件如图 5.15 所示。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



图 5.15　目前常见的比特币钱包软件

此外还有一种硬件钱包，具有极高的安全性。如由 Armory Technologies Inc. 公司开发的 Armory（军火库）钱包，它是一种开源的钱包软件，提供硬件冷存储及多重签名支持，号称“最好的、用于个人和机构的、存储比特币和私钥的钱包解决方案”。

TIPS:

所谓“冷存储”，又叫“冷钱包”，是指在一台不与网络连接的硬件设备（如一台不上网的笔记本电脑等）上存储比特币的钱包和私钥信息。由于不上网，所以黑客、恶意软件、病毒软件等无法盗取用户的比特币及私钥信息。

当需要支付和收取比特币，以及需要使用私钥的时候，Armory 钱包的工作过程如下：先使用离线钱包的“只读”备份来创建交易，这个备份具有与其他在线钱包相同的大部分功能，可以在线创建交易但不能签名，所以不能直接存取钱包中的比特币，只有使用离线的冷存储钱包才能签名；在创建交易后，将交易信息保存到 U 盘中，将 U 盘接入离线的电脑上，再使用离线钱包对 U 盘中的交易签名后更新 U 盘中的交易信息，最后将 U 盘接回在线的电脑上向网络广播交易信息，在得到网络的确认、完成交易后再导入离线的冷存储硬件中。由于安全性极高但使用过程复杂，所以 Armory 钱包一般只适用于拥有大量的比特币、对于安全性极度重视的用户，为了安全性而不得不牺牲一定的便捷性。

5.5.3　交易身份验证

尽管有不同的钱包系统用以确认和定位交易双方的身份，但是比特币系统

仅供非商业用途或交流学习使用

中交易双方身份机制的不足之处在于：由于其匿名特性，使其只与账户紧密关联，但不具备与实际身份的强有效关联，从而限制了交易的类型，只能用于比特币这样的数字货币的支付交易。而一旦区块链进入如公证、数字资产、保险、审计等更宽广的领域，则需要将虚拟的账户与个人在网络之外的实体身份有效链接起来，为这类交易的信任产生提供更坚固的基石。

目前已经有一些区块链应用聚焦于交易双方身份验证领域[①]，如 ShoCard 是一种将实体身份证件的数据指纹保存到区块链上的服务，它的原理是：用户用手机扫描自己的身份证件，ShoCard 应用会把证件信息加密后保存在用户本地，把数据指纹保存到区块链上。区块链上的数据指纹受一个私钥控制，只有持有私钥的用户自己才有权修改，ShoCard 也无权修改。同时，为了防范用户盗用他人身份证件扫描上传，ShoCard 还允许银行等机构对用户的身份进行背书，确保用户身份证件的真实性。

OneName 则提供了另一种身份服务，通过这一软件，任何比特币的用户都可以把自己的比特币地址和自己的姓名、Twitter、Facebook 等账号绑定，这相当于为用户的每个社交账户提供了一个公开的比特币地址和进行数字签名的能力。

中国有一家创业企业 IDHub 正在基于区块链技术建立去中心化的数字身份应用平台，它已经与广东省佛山市禅城区合作建立了 IMI 身份认证平台。用户可以通过 IMI 创建身份，经过有权限的组织或个人认证（认证采用有权限机构用私钥加密、第三方用公钥解密方式，为个人用户增加信用），可以授权第三方查询或使用。据介绍，用户下载移动客户端，通过私钥对个人身份数据进行加密后，完成上链注册，在日常生活中需要运用身份验证时，只有在客户授权相关方之后才能使用其身份信息，办理出生医学证明、学历、学位、无犯罪记录等 20 项公证业务不需要再去现场，委托买楼等 13 个事项可实现“1 小时出证”。该平台目前接入的认证单位包括公证处、银联和佛山市政府部门等。此外，还有数家医院接入平台，用户授权后可以实现医院之间对用户医疗信息的查询和共享，提高个人用户在享受医疗服务时的便捷度和信息安全程度。

① 参考中信出版社出版的《区块链：从数字货币到信用社会》一书。

仅供非商业用途或交流学习使用

5.6 区块链的造链机制

区块链技术的一项重要功能是把交易信息按时间序列保存到一个一个的区块上，然后向全网络的每个节点公开，由全体共识来确认，并且可追溯，最终实现了历史信息不可篡改。

在不同规模的区块链上，无论是公有链、联盟链还是私有链，交易发生后都必须向全网广播以获取全网络的验证和确认。此时，作为区块链网络的维护者（以比特币为例，我们称之为“矿工”），他们要做的事情就是实时监听网络广播，及时收集当前一段时间内的交易信息，然后进行验证，完成验证后就要打包到自己创建的区块内，实现区块链不断延长，这一过程可以被形象地称为造链流程。

5.6.1 验证接收信息

在比特币系统中，矿工随时通过挖矿软件监听网络中广播的交易信息内容。当某个节点向比特币网络发出了自己完成的交易信息后，矿工将会第一时间接收到，存放在内存池（Mempool）中，然后进行验证。

比特币作为一种数字货币，它不是单独存在的，而是与交易信息紧密关联的。在比特币系统中，无论多少数量的比特币，都必然有来源、有去向，无论是矿工成功挖矿所得到的奖励或收到的交易费用，还是普通用户通过购买取得的比特币或收到的比特币转账，交易中使用的比特币都有可以追溯的地址，既实现了层层交易的可追溯，也实现了对比特币的“双重支付”等问题的解决，可以杜绝虚假交易。

TIPS:

因此，作为矿工，对接收到的交易信息首先要做的验证是：交易中的比特币来自哪里？是否来自区块链上已经确认过的区块中封装的交易？这些比特币是未花费（Unspent）的状态，还是已花费（Spent）的状态？

这些交易信息采用脚本语言 Script 编写，包括“输入”和“输出”两方面的信息。在验证过程中，矿工会将交易信息逐项进行确认，对以下几个方面进行验证：在交易双方中，支付一方的比特币是否有合法的来源（来自之前的哪

仅供非商业用途或交流学习使用

个区块的哪笔交易），支付的金额有没有超过所拥有的比特币总额，是否采用了私钥加密签名，并用该用户的公钥来验证支付方是否拥有用于支付的比特币；同时也要确认接收一方的地址并验证接收方的公钥，验证该交易中的比特币是否“被支付过”，以避免“双花问题（同一笔资金被多次使用）”。当一名矿工对一项交易验证通过后，他会把交易打包到自己收集的交易包内，并向交易发起者发送一条信息，表示交易有效并被验证通过。如果这笔交易被矿工验证为无效，那么该矿工将拒绝接收，并给交易发起方发出一条信息，表示交易被拒绝。

以下是一个区块的内容[①]摘录：

```
In:
Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd
04470b9a6
Index: 0
scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446
618c4571d1090db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd
7d5d6cc8d25c6b241501
Out:
Value: 5000000000
scriptPubKey: OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d
35549d OP_EQUALVERIFY OP_CHECKSIG
```

上述 Script 脚本的含义如下。

1. 输入方（In）

（1）Previous tx：收入来源交易区块的哈希值，说明了即将支付的比特币来自以前的哪笔交易。它实际上是一个区块的地址，指向历史区块中的某一笔交易或多笔交易（如果用于支付的比特币来自多笔交易）。

（2）Index：指明是收入来源区块中具体哪笔交易的 Out 值，因为在前一笔交易中包含多笔交易的信息，通过索引值确定待支付的金额来自具体的哪笔交易。

（3）scriptSig：表示交易支付方对该交易的 ECDSA 签名认可，即支付者的私钥签名。

① 参考：http://8btc.com/article-2006-1.html。

仅供非商业用途或交流学习使用

2. 输出方（Out）

（1）Value：发送的币值，以“聪（Satoshi）”为单位，1 个比特币=100 000 000 聪（1 亿聪）。

（2）scriptPubKey：接收方的公钥脚本。

3. In 与 Out 的关系

总的原则是：每笔交易 Out 的总额必须小于或等于 In 的总额。但是，在某些区块的脚本里，只有输出方的金额，没有输入方的金额，这是因为该交易通过 In 的 Pervious 与 Index，追溯到之前某个区块中的某笔交易的 Out 值，从而获得 Value。

TIPS:

> 此外，在每次的支付过程中，如果出现找零的情况，则必须另外创建一个比特币地址来存放找回的零头，否则就会导致找回的零头成为创建区块的矿工的交易费用，被矿工收走，造成用户的损失。

矿工收取的交易费用基于用户自愿的原则，由输入与输出的比特币的差额来提供。如果用户用很大的输入支付了较小的输出，则必然有较大的差额（找零）。此时如果没有一个虚拟地址来存放这些零钱（相当于用户要为这些找零创造出一个虚拟的“输出”，比如指向自己的另一个钱包地址），那么这一差额就被系统认定为用户支付给创建区块的矿工的交易费用，进入矿工的钱包。

下面用一个通俗的例子来说明“流入”和“流出”的数量关系。

（1）小张有 10 个比特币，是通过一次购买交易获得的，现在送给朋友小王 10 个比特币，在这种情况下，区块的脚本语言中只有一个流入和一个流出。

（2）小张有 10 个比特币，是通过之前的某次交易获得的，现在送给朋友小王 5 个比特币，在这种情况下，区块中会有一个流入，但是会有两个流出，其中一个流出指向送给小王的 5 个比特币，另一个流出指向小张自己，因为小张要得到找零的 5 个比特币。

（3）小张有 10 个比特币，是通过之前的两次交易获得的，现在要送给朋友小王 10 个比特币，在这种情况下，区块中会有两个流入和一个流出，其中的两

仅供非商业用途或交流学习使用

个流入分别指向之前的两次交易。

（4）小张有 10 个比特币，是通过之前的两次交易获得的，其中一次获得了 6 个比特币，另一次获得了 4 个比特币，现在要送给朋友小王 7 个比特币，在这种情况下，区块中会有两个流入，分别指向前两次交易的 6 个和 4 个比特币，以及两个流出，分别指向给小王的 7 个比特币和找零的 3 个比特币。

5.6.2　创建区块

在完成了上述验证后，矿工认定，这一交易信息是一份正常的交易，于是将这笔交易存储到自己计算的区块的交易组内。经过一个特定的时间长度（约 10 分钟）后，矿工把在当前这一时段内接收到的所有交易信息作为一个数据包，开始进行默克树计算，目的是计算出自己将要创建的区块中的交易信息默克根，并把这个默克根写入区块头中，代表本区块中的所有交易信息。

TIPS:

默克根的作用很重要，它是对全部信息的逐级哈希浓缩，最终形成了区块中全部交易的汇总值。在全网络的节点进行区块链的历史信息验证时，通过对比默克根就可以快速发现交易信息是否发生了篡改。

至此，矿工完成了交易信息的封装流程，接下来就要开展铸币流程了。

在比特币系统中，对于矿工来说，随着挖矿人数越来越多，而能够挖到矿、获得构造区块机会的永远只能有一个，所以机会变得越来越稀缺、竞争变得越来越激烈。铸币流程的关键就是：天下武功，唯快不破！也就是说，在铸币流程中，比拼的就是速度，实质上是比拼算力的大小。

1. 交易信息到达矿工节点

每时每刻，在比特币网络上都有无数的节点在监听着网络中的交易广播，根据规则，在大约 10 分钟的时间段内，数以千计的交易被一个节点收集下来，打包到一起，然后开始“铸币”，也就是挖矿。挖矿成功的矿工获得了比特币的奖励，增加了比特币货币的供应，铸币成功；同时，矿工还维护着比特币系统的安全，通过验证交易、传递交易，矿工承担了防止欺诈交易、避免“双重支付”的责任。

仅供非商业用途或交流学习使用

一笔比特币交易被用户广播到比特币网络中，由于比特币网络是一个点对点网络，因此最近的节点首先收听到这一信息，验证通过后由这个节点再向其他节点广播。与此同时，在用户网络接入点附近的节点并不是唯一的，其他的节点也在进行“验证—广播”的工作，如果这是一笔合法的交易，那么很快就会“一传十、十传百”地在网络中传播，在几秒之内就可以传播到网络中的全部节点，其中就包含了大量的矿工节点。矿工节点除进行“验证—广播”工作外，还收集这些信息，随时准备进行挖矿。

TIPS:

如果未通过验证，则第一批收到交易信息的节点会向用户返回“未通过验证”的信息，并不再向其他节点广播这笔交易。所以，基于比特币设计好的私钥和公钥加密规则，客观上各节点在比特币网络中也起到了过滤垃圾信息的作用。

区块内的每笔交易都有可以追溯的来源，同时又为下一笔交易（本次交易的输出）提供了去向，所以下一笔交易也是可以追溯的。

TIPS:

但是，由于交易是在全球范围内的比特币系统上随机发生的，所以有时候包含父交易信息的区块的传播速度落后于包含子交易信息的区块的传播速度，从而导致区块中存在找不到来源的子交易，这种交易被称为孤立交易。

在这种情况下，接收到交易信息的节点会将孤立交易存放到一个临时交易池中，一边等待接收它的父交易信息，一边向其他的节点进行广播。一旦接收到了父交易信息，有关的孤立交易就会从节点的临时交易池中释放出来重新验证和链接，并再次向网络发送广播，使这一交易成为合法交易，被矿工节点收集并纳入区块所创建的交易包中。

当然，比特币系统设立了规则，规定了内存中存储的孤立交易数量的最大值，即客户端源代码中的 MAX_ORPHAN_TRANSACTIONS。当孤立交易数量达到 MAX_ORPHAN_TRANSACTIONS 时，系统就会随机挑选一个或多个孤立交易扔掉，以保证临时交易池的大小不超过限制。

仅供非商业用途或交流学习使用

交易信息在到达矿工节点之前的流程如图 5.16 所示。

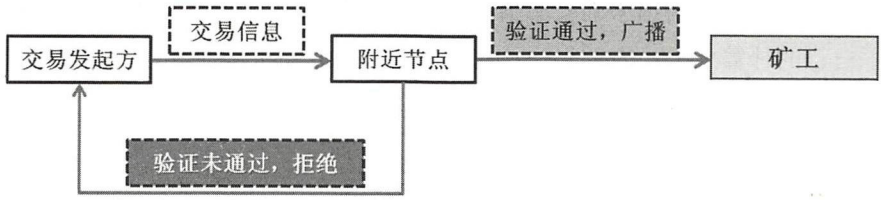


图 5.16　交易信息在到达矿工节点之前的流程

2. 矿工将交易信息打包和铸币

接下来，想象一下我们正在观察比特币系统中的一名矿工，假设他的虚拟名称是王五，他也许有一个矿场，也许是一个矿池的参与者，因为在目前的算力状况下，单机的矿工已经没有希望挖到比特币了。

矿工王五在接收到交易信息后，自己也要进行交易验证，验证通过后，在将交易信息转发的同时，也将这笔交易信息存入自己计算区块的交易包中，并继续接收下一笔交易信息。

在上一个区块创立的时间点后，矿工王五从接收到第一笔交易信息开始，就建立了一个备用区块，这个备用区块不断地装入接收到的交易信息，并且将收集到的交易信息进行快速的默克树计算，目的是获得最终的默克根，因为这个默克根将用于构造区块头的数据结构。当然，同时加入区块头的还有父区块的哈希值、本区块的时间戳（采用 UNIX 纪元时间的编码，即从 1970 年 1 月 1 日 0 点开始一直计算到当前时间的总秒数）。

另外，他的矿机在专用软件的驱动下开始飞速运转，根据系统给定的目标值，通过不断变动 Nonce（随机值）放进区块头，一个又一个的区块哈希值被飞快地计算出来，并与系统给定的要求（例如，哈希值的前 20 位均为 0）进行对比，不符合的数据被抛弃，马上切换到下一个数据……在矿工王五的矿场里，看不见的比特流在一排排矿机间飞快流动，使得矿机的温度迅速上升，巨大的冷却风扇适时启动，带走矿机发出的热量；同时，来自供电系统的电流源源不断流入，为矿机提供了充足的动力。而这一切资源的投入换来的是屏幕上飞速闪动的哈希值字符串。1 秒，2 秒……随着时间的流逝，矿工王五计算出的哈希值前面的 0 越来越多，终于，他计算出了前 20 位为 0 的哈希值，此时距离开始

仅供非商业用途或交流学习使用

计算已经过去了近 11 分钟。矿工王五看了一下系统接收到的信息，目前尚未收到其他矿工节点广播出来的哈希值碰撞成功的信息，于是他很高兴地向系统广播了自己的计算成果，表明自己构造的备用区块成为正式的区块，然后等待其他矿工的确认。

一般情况下，这种确认只是一种程序性的过程，因为他的矿机已经无数次完成了类似的过程，计算过程不会有任何问题，只要计算速度超过其他的矿工，就意味着他已经成功地抢到了此次“算力竞争”的第一名，他创建区块的第一笔交易——给自己 12.5 个比特币奖励的“币基交易”将会得到所有人的确认，以及得到交易包中所有交易提供的交易费用之和（以比特币计价），总之这是一个收获颇丰的过程。不久，矿工王五收到了不同节点的确认信息，他的区块创建成功，成为比特币区块链上最新的“父区块”。

在等待确认的同时，矿工王五也没有闲着，他会继续监听和接收网络中的交易信息，并以自己刚刚创建出来的区块为上一级区块，进行下一个区块的创建。区块就是这样源源不断地被创建出来的，比特币也随之不断地产生，这就是比特币网络的“铸币”过程。

TIPS:

> 根据中本聪建立的规则，比特币的奖励每 4 年减半，从最初的一个区块奖励 50 个比特币，然后减半到 25 个比特币，目前已经减少到 12.5 个比特币。按照这个规则，预计到 2140 年，区块的创建就不会再有奖励的比特币发放，因为总量 2100 万个比特币已经“铸造”完成。

当然，也有很多的时候，正当矿工王五的矿场在飞速计算时，他收到了来自地球上某个角落的另一名矿工向全网广播的“解题答案”，并且经过检验发现这个答案是正确的，这意味着这一轮“算力比拼”他没有得到第一，区块由那名矿工创建，这 10 分钟内所有的收益也与他无关了，而他付出的电力、矿机甚至人力的成本，也成为没有收益的沉没成本。矿工王五能做的就是迅速转向，他将以新的区块作为“父区块”，参与下一轮的“算力竞争”。

3. 交易的费用

参与铸币的矿工节点与普通的用户有两点最大的不同：一方面，矿工会不

仅供非商业用途或交流学习使用

断地增加经济投入以提高算力，以取得在算力竞争中的胜利；另一方面，矿工在本地会随时对从比特币创立以来到当前的最新、最长的区块链进行副本备份，以便能够随时掌握区块链的动态，为自己创建区块链提供良好的条件。因此，从经济和人性的角度来看，矿工作为比特币区块链的维护者是有较大的投入的，他们需要获得一定的收益才能有持续不断进行区块创建的动力。

而对于整个区块链来说，只有保持区块的不断创建才能使区块链不断地对全网络的交易信息进行确认和封存，保证区块链系统得以稳定、可靠地运行。但是，随着算力竞争的加剧，以及区块创建奖励减半规则的实施，矿工的来自区块奖励的比特币收益逐步减少，交易费用将成为矿工维护区块链、进行交易确认、不断创建区块的合理收益所在。

TIPS:

交易费用的存在也有助于减少欺诈交易等现象，因为对于交易发起方来说，如果造假就需要先付出一定的成本，从而减少了一些人意图进行欺诈交易的想法；反之，如果系统对交易的要求都是免费的，则意味着造假者没有任何成本，必将大大增加欺诈交易的数量。

交易发起方都希望自己的交易能够尽快被矿工确认，尽快被收录到区块中锁定，但是在矿工创建区块的时间段内，他会收到成千上万的交易信息，先确认哪项交易信息？显然，矿工从自己的收益出发，会按照交易费用的高低，优先确认交易费用高的交易信息。因此，如果交易发起方希望能够尽早完成交易，则都会根据自己交易的轻重缓急，在交易中设置一定的交易费用。

比特币系统也有“最小交易费用”的规则。交易费用的计算不是基于交易金额（比特币价值）的大小，而是基于交易信息的体积。目前最小交易费用规定为每千字节 0.0001 个比特币（或每千字节万分之一个比特币）。交易费用可以由钱包软件按当前规则自行计算和设置最小交易费用，也可以由交易发起方在脚本中自行设定。当交易发起方自行设定时，其设定方法是：

交易费用 = 输入地址的比特币总和 − 输出地址的比特币总和

用户是通过客户端（钱包软件）来进行交易的，钱包软件中存储的都是不同来源的比特币地址，对应着不同的比特币数量。

TIPS:

因此，在一次交易中，存在多个输入来源（比特币地址），每个输入来源都包含着相应数量的比特币，钱包软件会根据需要支付的比特币总量，选择使用哪些输入来源来进行支付组合，并计算出用于支付的比特币总量。

例如，共有 15.03 个比特币，而交易的输出方可能是一个，也可能是几个，由钱包软件根据交易者的要求，设定给每个输出地址多少个比特币，全部输出对应的比特币之和是本次交易支付的比特币总量，比如是 15 个比特币，那么总输入和总输出二者之差为 15.03−15=0.03 个比特币，这 0.03 个比特币没有对应的钱包地址，这就是交易发起方给矿工的交易费用，将由矿工收集。如果交易发起方没有进行这样的安排，钱包软件就会根据交易的字节数，按“每千字节万分之一个比特币”的比例自行计算最小交易费用，自动组合好输入和输出。

交易方也可以自行设定免费的交易，即在交易脚本中设定总的输入比特币与总的输出比特币相等，则这笔交易就没有留给矿工的交易费用了。这样的交易在矿工的交易池中会处在最低的优先级，也许会在经过几个区块的创建后，随着币龄的增加，被某些矿工出于增加区块币龄的考虑而确认，也有可能会由于较长时间得不到确认而随着矿工内存的刷新而丢失。所以，如果是比较大额的交易，则应该提高交易费用。

另外，需要注意的是，在一些用比特币支付的交易中，存在着较大的找零情况，如果交易支付方忽视了给找零的比特币设定一个自己的钱包地址，那么这些比特币不会回到交易支付方的钱包软件中，会被矿工按交易费用收集，同时会给予本笔交易很高的确认优先级。虽然交易得到了尽快的确认，但是显然这并非交易支付方的本意。

5.7 区块数据解读

区块链技术目前最成熟的应用是比特币，自 2009 年上线运行以来，系统没有发生过故障或死机。本章我们介绍区块链的数据结构，也是以比特币作为区块链的典型代表的，同时介绍了数据的加密和压缩过程，以及区块链中用到的

各类算法。接下来我们仍将以比特币为例，进入比特币的区块内部，了解区块链的基本构成单位——区块的数据结构。目前存在的其他区块链，如以太坊，其区块内部的数据情况与比特币大同小异。

TIPS:

基于去中心化的设计，并没有一台中心服务器来存放区块数据。作为一条完整的区块链，区块数据存放在每个节点本地的存储设备中，全网所有的节点共同维护这一分布式账本，以保证账本的完整、健康和可靠。

当我们使用比特币客户端连接比特币网络时，系统首先要做的事就是与其他节点的分布式数据库同步这条区块链中的数据。以常见的比特币客户端 Bitcoin-Qt 为例，区块数据就存储在电脑中 Bitcoin-Qt 的相应文件夹下，以“blocks”命名该次级文件夹[①]。

打开 blocks 文件夹，其中存在许多以 blk***.bat 命名的文件，每个文件就是一个区块数据，如图 5.17 所示。

Index of /home/simone/.bitcoin/blocks/

Name	Size	Date Modified
[parent directory]		
index/		6/6/13 9:18:34 AM
blk00000.dat	128 MB	5/11/13 4:40:17 PM
blk00001.dat	128 MB	5/11/13 4:46:37 PM
blk00002.dat	128 MB	5/11/13 4:53:17 PM
blk00003.dat	128 MB	5/11/13 5:11:23 PM
blk00004.dat	128 MB	5/11/13 5:15:28 PM
blk00005.dat	128 MB	5/11/13 5:24:26 PM
blk00006.dat	128 MB	5/11/13 5:30:41 PM
blk00007.dat	128 MB	5/11/13 5:37:08 PM
blk00008.dat	128 MB	5/11/13 5:43:04 PM
blk00009.dat	128 MB	5/11/13 5:48:32 PM
blk00010.dat	128 MB	5/11/13 5:57:58 PM
blk00011.dat	128 MB	5/11/13 6:12:10 PM
blk00012.dat	128 MB	5/11/13 6:27:48 PM

图 5.17　blocks 文件夹中存放的区块数据文件

这些 blk***.dat 文件可以使用十六进制文件查看器 hexdump 打开。例如，用 hexdump 打开创世区块的内容，如图 5.18 所示。

① 参考：http://p2pbucks.com/?p=4242。

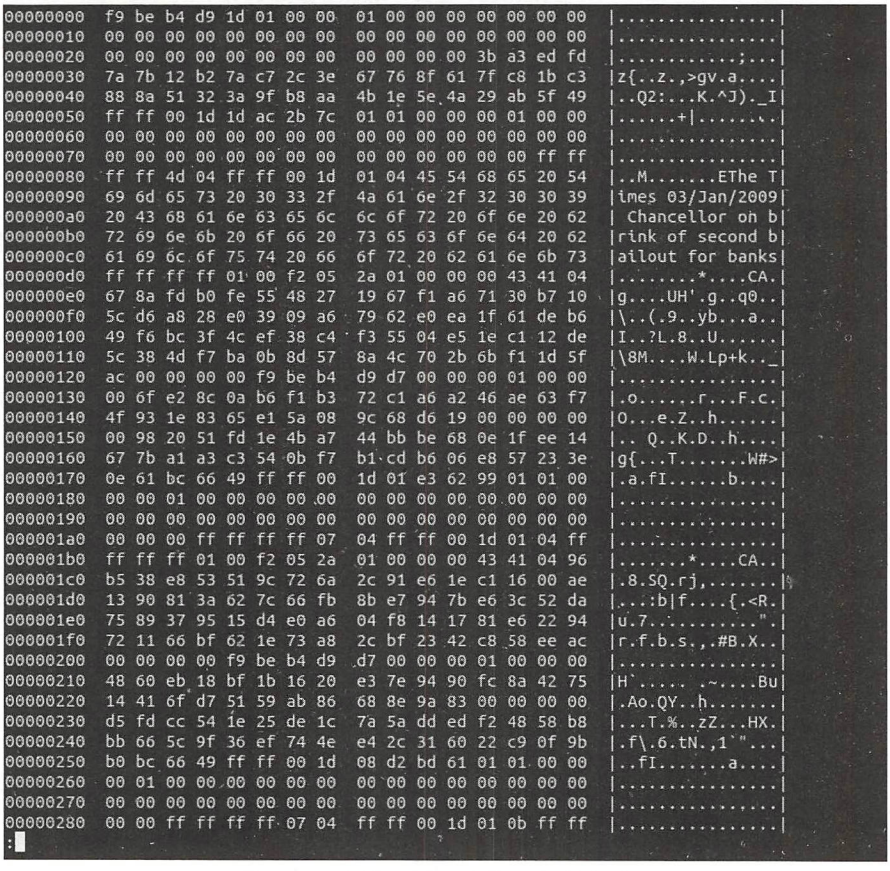


图 5.18　创世区块的内容（十六进制）

其结构分为以下 5 个部分。

（1）神奇数：是一个固定值 0xD9B4BEF9，其作用是作为区块之间的分隔符。

（2）表示当前区块大小的数据：4 字节。

（3）当前区块的区块头信息：80 字节。

（4）交易计数：1～9 字节，记录本区块中交易的数量。

（5）交易详情：记录和保存本区块中写入的交易细节，没有固定的大小。

一个完整的区块数据组成结构如表 5.9 所示。

表 5.9　一个完整的区块数据组成结构

标　记	字节	含　义	备　注
BlockChain	区块链		
Magic Number	4	神奇数	0xD9B4BEF9
Block Size	4	区块大小	
Block Head	80	区块头	
Block Body	-	区块体	
Block Head	区块头		
Version	4	区块头版本号	
Prev Block Hash	32	前一区块头哈希值	Hash256(x)= SHA 256(SHA256(x))
Merkel Root Hash	32	交易内容 Hash256	
Time	4	UNIX 时间戳	从 1970 年 1 月 1 日 0 时起的秒数
Bits	4	目标值	用来标注挖矿难度
Nonce	4	随机数	用来调整当前区块头 Hash
Block Body	区块体		
Transactions Counter	1～9	交易数量	Variable Interger 类型
Merkle Root	-	默克根	
Merkle Root	交易内容		
Transaction 1	-	交易 1	挖矿奖励（Coinbase）
Transaction 2	-	交易 2	
……	……	……	
Transaction n	-	交易 n	
Transactions	交易单		
Version	4	交易单版本	
Inputs Counter	1～9	收入单数量	Variable Interger 类型
Inputs Detail	-	收入单内容	
Outputs Counter	1～9	支出单数量	Variable Interger 类型
Outputs Detail	-	支出单内容	
Lock Time	4	锁定时间	从当前时间起无法用于支付的秒数
Inputs Detail	收入单内容		
Inputs 1	-	收入单 1	
Inputs 2	-	收入单 2	
……	……	……	
Inputs n	-	收入单 n	

续表

标　记	字节	含　义	备　注
Outputs Detail	支出单内容		
Outputs 1	-	支出单 1	
Outputs 2	-	支出单 2	
……	……	……	
Outputs *n*	-	支出单 *n*	
Inputs	收入单		
Previous tx Hash	32	前一交易单 Hash	
Previous Output Index	4	引用交易单支出单索引号	
Inputs Script Length	1～9	收入脚本长度	Variable Interger 类型
Inputs Script	-	收入脚本	
Sequence Number	4	序列号	0xFFFFFFFF
Outputs	支出单		
Amount	8	比特币数量	单位：1 聪=0.00000001 比特币
Outputs Script Length	1～9	支出脚本长度	Variable Interger 类型
Outputs Script	-	支出脚本	
Inputs Script(coinbase)	收入脚本（挖矿）		
Signature Length	1～9	公钥长度	0x48
Signature	72	签名	
Outputs Script(coinbase)	支出脚本（挖矿）		
Pubkey Length	1～9	公钥长度	0x41
Pubkey	65	公钥	
OP_CHECKSIG	1	操作符（检查签名）	0xAC
Inputs Script(Standard)	收入脚本（标准）		
Signature Length	1～9	签名长度	0x48
Signature	72	签名	
Pubkey Length	1～9	公钥长度	0x41
Pubkey	65	公钥	
Outputs Script(Standard)	支出脚本（标准）		
OP DUP	1	操作符（复制堆栈）	0x76
OP_HASH160	1	操作符（Hash160）	0xA9
Pubkey Hash Length	1～9	公钥 Hash160 长度	0x14

续表

标　记	字节	含　义	备　注
Pubkey Hash	20	公钥 Hash160	Hash160(x)=Ripemd160(SHA256(x))
OP EQUALVERIFY	1	操作符（检查相等）	0x87
OP CHECKSIG	1	操作符（检查签名）	0xAC

不过，对于普通用户来说，由于并不熟悉十六进制代码，因此用 hexdump 查看起来相当吃力。所以现在有很多第三方软件提供了对于比特币区块的浏览功能，比如 Blockchain.info 网站。用户登录该网站，可以查看中文界面的比特币区块的主要信息，相当方便。如图 5.19 所示为登录该网站的界面。



图 5.19　区块查看网站 Blockchain.info 登录界面

进入网站后，单击左侧的区块高度值，或在搜索框内输入区块对应的高度值、区块哈希值，就可以打开相应的区块，看到主要的信息。如单击区块号（高度值 516022），即进入区块 516022 的详细界面，如图 5.20 所示。

首先是“概览”部分，显示了区块高度（区块号）、本区块的哈希值（也就是由矿工计算出来的满足难度目标值要求的哈希值）、前后区块的哈希值，以及时间戳（已经转换为现实时间）、矿工（播报方，本区块为 BTCC Pool，即比特大陆矿池）、难度系数等与区块创建有关的参数；其次还展示了本区块的体积大小（单位为千字节）、区块重量、版本号、区块中包含的交易数量、比特币总输出量、比特币总交易量、默克根的值、创建区块所用到的随机值，以及得到的

奖励 12.5 个比特币、总交易费用 0.08163773 个比特币，按当天的比特币价格约 7000 美元计算，交易费用超过 560 美元。

区块 #516022

概览	
交易次数	544
总输出量	833.181112 BTC
预计交易量	91.37955507 BTC
交易费	0.08163773 BTC
高度	516022 (主链)
时间戳	2018-03-31 20:59:35
时间	2018-03-31 20:59:35
播报方	BTCC Pool
难度系数	3,462,542,391,191.56
计算目标	391203401
大小	237.819 kB
重量	822.792 kWU
版本	0x20000000
随机数	3145690442
新区块奖励	12.5 BTC

哈希值	
哈希值	00000000000000000011c79cb8501bfa83eeed28cdedb0fb6ef292394c6e2932
上一区块	00000000000000000033b677e310097c4a55dad9b71eab0f9f2eb519bb3b71c
下一区块	00000000000000000086a68c5d4a7faa840252121df6ade497a126a087d291a
二进制哈希树根	45c88ae9968313afb8819cb9b86725140dfdd3cb4b7c198c97218a6489aa9bf6



图 5.20　区块 516022 的详细界面

其下是交易记录的详情，如图 5.21 所示。

Transactions

3bca55dd9f38813976e618f639554d1617a9c8abf747a97c0794355ad093cc2e			2018-03-31 20:59:35
No Inputs (Newly Generated Coins)	➡	13TEThZNnKPk34HYAuo1QqYMwDdjF3qeHx Unable to decode output address	12.58163773 BTC 0 BTC 12.58163773 BTC
8380efa412c7746a79a8f152af47a2f575b8a09f76532b4964430093c78ca5fb			2018-03-31 20:55:01
19S7iSKhy1dPJ3hTdSEjDwPkFY8nWFs8dj 1HaZCpvkrWGReaZgsT6hntBK9WsVrKhA6M	➡	1BnssZi5i2iTETRtg168DzHjpNCeoWZodZ 1KQPx6dvgiYbAbq7vGGFSgqK6fLv25UHNj	0.02233034 BTC 0.00979566 BTC 0.032126 BTC
160c551d9119e28abc1cd93065dc87617ecf6ca9e3e19700fa47bd00d3aacccd			2018-03-31 20:57:28
1K7vTgtkF2Q44oFpZzHsshE799rPNKoePe	➡	15uzk5JKLfqm6Ruk7zYHfjEoN9Y9RmaPBs 1B38srTndaKBGcPmjX31avpTefyt4A4CjH	0.01410938 BTC 0.001 BTC 0.01510938 BTC

图 5.21　区块 516022 中的交易记录详情（部分）

图 5.21 摘录了部分交易记录，从中可以看出，第一笔交易是币基交易，就是获得区块创建的 12.5 个比特币奖励，以及交易费用的 0.08163773 个比特币，所以没有输入信息，只有输出信息，输出地址指向矿工的钱包，总的比特币数量为 12.581637773 个比特币。

从第一笔交易开始，显示区块中每笔交易的记录，包括交易编号、输入地址、输出地址、每笔交易的比特币数量、交易比特币总量。如果单击交易编号（左上角的一串编码），则可以看到该笔交易的详情。如单击第二笔交易的编号，可以看到如图 5.22 所示的内容。



图 5.22　区块 516022 中第二笔交易的概况

从图 5.22 中可以看到交易信息的大小为 372 字节、重量（本交易占区块总体积 1MB 的比例，以交易体积占区块重量单元的个数来表示，因为每重量单元为 1MB 的 1/4000000 份，即 1 个重量单元=1MB/4000000=0.25 字节，所以 372 字节的交易占据的重量单元数=372/0.25=1488，也可简单计算为字节数×4），还可以看到矿工接收交易的时间、位于哪个区块内、取得节点确认的数量，以

及输入、输出比特币的总量，二者的差额就是本次交易的交易费用 0.00224774 个比特币。

另外，还可以看到输入脚本和输出脚本，用于其他节点对本次交易进行验证。

如果单击输入地址，则可以进一步向上追溯，查看交易支付方的比特币的交易来源，以及是否确认、是否已经使用等信息。由于涉及层层溯源，并且对其信息的解读基本相似，所以读者可以自行了解，此处不再赘述了。

chapter SIX

第6章

以太坊

随着区块链技术的快速发展，除比特币网络成为全球成熟的区块链应用以外，另一个在全球范围内发展和成熟、拥有大量用户、成为底层开发平台的区块链就是以太坊（Ethereum），它也是除比特币外另一条目前极为成熟的公有链。从理念上来说，以太坊突破了比特币对区块链应用的限制，让人们对区块链的认识不再局限于数字货币，将应用领域通过智能合约的形式进一步拓展到各行各业。不同于比特币网络的封闭和单一，以太坊在比特币的基础上发展起来，总结了比特币自创立以来在实际应用中的优势和不足，提高了区块链技术应用于社会生活的特性，更加开放和兼容，成功地拓展了区块链技术的边界。

6.1 以太坊概述

以太坊，英文名为 Ethereum，它是一个平台和一种编程语言，如果说比特币相当于区块链 1.0，那么以太坊就是区块链 2.0，它使开发人员能够建立和发布下一代分布式应用。以太坊可以用来编程、处置、担保和交易任何事物，如投票、域名、金融交易所、众筹、公司管理、合同和大部分的合约、知识产权，以及得益于硬件集成的智能资产。它包括一套内置的、图灵完备（指计算能力强大，满足任意数据按照一定的顺序可以计算出结果）的脚本语言，允许通过智能合约来编写代码，每个合约相当于一个自动代理，当它接收到一项特定的

输入交易后，合约就会运行特定的代码，使交易内容得以发生，进一步增强了去中心化，高级的合约甚至能修改自身的代码。

总的来说，以太坊是以智能合约为主要特色功能的一种分布式计算平台，它提供了一种分布式虚拟机——以太坊虚拟机（Ethereum Virtual Machine，EVM），这种虚拟机采用系统内含的加密货币——以太币，可以执行点对点合约。

TIPS:

以太坊并不是以创造某种“币”为最终目的的，而是致力于成为一个优秀的底层合约，在其基础上可以创建任意高级的合约、货币及其他去中心化应用。这种特点使以太坊成为一个平台而不是一种单一的应用。在以太坊平台上，有创意的开发者将不再需要启动他们自己的区块链，他们可以简单地用以太坊脚本编码实现他们的想法。简而言之，以太坊是一个去中心化的创新基础平台。

在以太坊出现之前，开发区块链应用通常采用的方法是：复制一份比特币源代码，然后通过修改底层代码如加密算法、共识机制、网络协议等，开发出一套新的区块链应用。比如，在比特币之后出现的很多山寨币就是以这种方式将比特币的开放源代码修改后开发出来的。以太坊平台转换了思维模式，它对底层区块链技术进行了封装，让区块链应用开发者可以直接在以太坊平台上进行开发，调用以太坊的底层功能模块实现区块链的种种功能，使开发者将注意力集中于应用产品本身的开发，无须研究底层的实现过程，从而大大降低了区块链应用的开发难度。

6.2 以太坊的创立和发展

以太坊从创立到成为全球性的公有链之一，仅有短短数年。2013 年 11 月，来自俄罗斯的 Vitalik Buterin 发布了一份白皮书，正式提出了以太坊的概念和基本代码的创建。Vitalik Buterin 当时才 20 岁，可以称得上“科技神童”。他既是一名程序员和作家，也是一名加密货币的研究者。除了是以太坊的创始人，他

还是《比特币杂志》（*Bitcoin Magazine*）的联合创始人。Vitalik Buterin 出生在莫斯科，6 岁时随父母移民到加拿大。

2013 年年末，Vitalik Buterin 为了建立去中心化的应用程序，与比特币的核心开发人员进行了交流。他认为平台需要更稳健的脚本语言来开发应用程序，但没有得到比特币开发人员的认同，于是他提出要开发一个有着更通用的脚本语言的新平台，这就是以太坊（Ethereum）最初的理念。2014 年 1 月，在美国迈阿密举办的“北美比特币大会”上，Vitalik Buterin 正式公开了以太坊项目。最初的开发团队成员除 Vitalik Butrein 外，还有 Mihai Alisie、Anthony Di Iorio 及 Charles Hoskinson；在迈阿密会议后，Gavin Wood 和 Jeffrey Wilcke 也加入以太坊的开发团队中来。以太坊的正式软件开发是在 2014 年年初通过一家瑞士公司 Ethereum Switzerland GmbH（Ethereum Suisse）进行的。2014 年 4 月，Gavin Wood 发表了《以太坊黄皮书》，在其中制定了以太坊虚拟机的技术规范。2014 年 6 月，在瑞士成立了非营利组织“以太坊基金会（Ethereum Fundation）”，这个基金会由行业参与者组成，目的是确保以太坊架构能够公平和独立地运作。

以太坊进入大规模的开发工作阶段是从 2014 年 7—8 月的一次线上众筹活动（实际上是预售以太币）开始的。当时为了整合开发人员、矿工、投资人及其他利益相关者的资源，以太坊开展了一次预售活动，要求参与者使用比特币来购买以太坊的加密货币——以太币（Ether）。2014 年 9 月，以太坊结束了为期 42 天的以太币预售，共筹集到 31 591 个比特币，购买了 60 102 216 个以太币，当时价值约 1800 万美元。这也是全球范围内在区块链领域的首次 ICO 行为。

TIPS:

众筹结束后，以太坊的开发进入了快速发展阶段，项目团队不断地在开发社区上推出测试版本，即 Proof of Concept（PoC），吸引了越来越多的人加入开发社区进行讨论。

以太坊各个版本的软件发布过程如下。

2015 年 5 月，以太坊发布原型代码，代号“奥林匹克（Olympic）”。这是公开测试版，给用户提供了 25 000 个以太币的找 Bug 奖励。

2015 年 7 月 30 日，以太坊发布试验性平台“前沿（Frontier）”版。这个版本只有命令行，只适合专业用户（开发者）使用。在 Frontier 版发布以后，开发人员开始在以太坊的活动网络上编写智能合约和去中心化的应用程序（App），矿工开始进入以太坊网络维护网络安全并挖矿以获得以太币。

2016 年 3 月 14 日，以太坊发布“家园（Homestead）”版。这是第一版稳定的以太坊，包含交易流程、燃料定价和安全方面的改进，影响到平台的激励机制。

2017 年 3 月，30 家企业和机构参与组建了非营利组织“以太坊企业家联盟”，参与者中既包括初创企业和研究机构，也包括世界 500 强企业；到 5 月份的时候，这个组织的成员增加到 116 个，其中企业成员包括区块链公司 Consensys、微软、三星、丰田研究院、芯片巨头 Intel、JP 摩根、德勤、埃森哲、加拿大国家银行等；到 7 月份的时候，成员数量超过 150 个，连万事达卡、思科公司也加入进来。加入联盟的成员数量多、分布广，也充分说明以太坊这一公有链所具有的吸引力。

2017 年 10 月 16 日，以太坊发布“大都会（Metropolis）第一部分：拜占庭”，其变化包括减少了 EVM 的复杂性，为智能合约开发人员提供了更多的灵活性等。

TIPS:

以太坊网络也提供了加密货币，它是一种系统内置的货币——以太币。在以太坊中，以太币扮演双重角色：一方面，以太币可以进行交易，可以与其他各种数字货币进行兑换，为各种数字资产交易提供主要的流动性；另一方面，在以太坊网络中进行交易需要支付交易费用，而以太币就是支付交易费用的一种机制。

以太币的数量单位有伟、萨博、芬尼、以太。其中，“以太”被用作普通交易，“芬尼”用来进行微交易，“萨博”和“伟”用来进行关于费用和合约实施的讨论。以“伟（wei）”作为基础单位，各数量单位之间的关系如表 6.1 所示。

表 6.1　以太币单位换算表

单　位	伟　值
伟	1wei
千伟（Kwei）	1000wei
兆伟（Mwei）	1 000 000wei
吉伟（Gwei）	1 000 000 000wei
microether 微以（萨博）	1 000 000 000 000wei
milliether 毫以（芬尼）	1 000 000 000 000 000wei
1 以太	1 000 000 000 000 000 000wei

自 2017 年以来，以太币的价格开始不断上涨，最高值超过 1200 美元，进入 2018 年后逐步回落，目前以太币的价格在 400 美元左右波动，总市值约为 390 亿美元。如图 6.1 所示为自 2015 年以太币上线以来的价格与市值走势图。

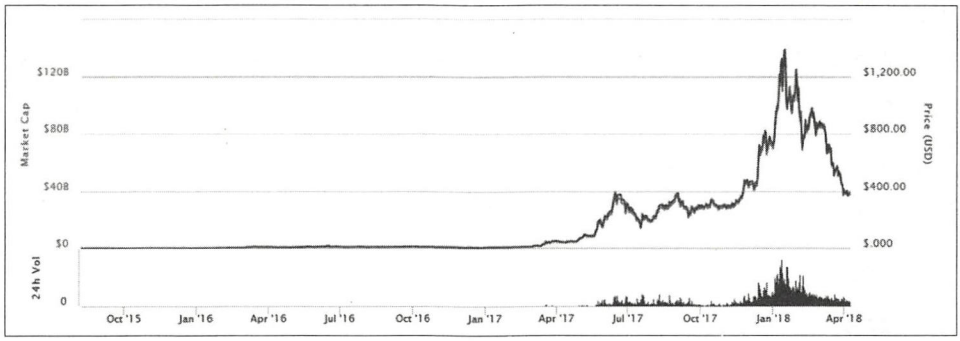


图 6.1　自 2015 年以太币上线以来的价格与市值走势图（左轴为市值，右轴为价格）

6.3　以太坊技术原理

6.3.1　以太坊与比特币的联系

以太坊是一种可编程的区块链，而不是给用户一套事先规定好的操作（比如，比特币交易），允许用户创建他们自己的、符合他们需要的、具备不同复杂程度的操作。从这种意义上来说，以太坊是一个平台，可以服务于多种不同的去中心化区块链应用，包括但不限于数字货币。

从狭义上来说，“以太坊”是指定义了去中心化应用平台的一套合约，它的核心是“以太坊虚拟机”，这个虚拟机可以执行具有任意算法复杂性的代码。用计算机术语来说，以太坊是“图灵完备”的，开发人员可以用现有编程语言（如 JavaScript 和 Python 等）的模仿语言在以太坊虚拟机上创建应用程序。

TIPS:

和其他区块链一样，以太坊也是建立在点对点的分布式网络基础上的，以太坊数据库也是由连接到网络上的许多节点共同维护和更新的，每个节点都运行以太坊虚拟机，执行相同的指令，因此有时候也被称为“世界计算机”。

以太坊和其他编程语言相似，本身是价值中性的，它的开发和使用，以及怎样产生价值，要取决于企业家和开发人员，但是某些应用相比其他应用在采用以太坊网络之后可以产生更大的收益，包括促进节点之间自动直接交互的应用，或者促进跨网络的团体协调活动的应用，比如，点对点的市场协作，或者复杂金融合约的自动执行等。比特币允许个人之间直接进行货币交换，无须任何中间人（如金融机构、银行、政府等）；而以太坊的影响更加深远，理论上，任何复杂程度的金融交易或交换都可以用运行在以太坊上的代码自动而且可靠地执行。

TIPS:

除了金融类的应用，任何对信任、安全、可靠的环境存在强烈需求的业务，比如资产登记、投票、管治、物联网等，都可以利用以太坊平台的技术特点取得新的突破。

以太坊的目的是基于脚本、竞争币和链上元协议（on-chain meta-protocol）的概念进行整合和提高，使得开发者能够创建任意基于共识的、可扩展的、标准化的、特性完备的、易于开发的和协同的应用。以太坊通过建立终极的抽象的基础层（其中内置有图灵完备编程语言的区块链），使得任何人都能够创建合约和去中心化应用，并在其中设立他们自由定义的所有权规则、交易方式和状态转换函数，因此，以太坊的开发功能相当强大，系统所用数字货币的主体框架简洁到只需要两行代码就可以实现，而其他合约（如信誉系统等）也只需要

不到 20 行代码就可以实现。智能合约也在以太坊的平台上创建，并且因为图灵完备性、价值意识（value-awareness）、区块链意识（blockchain-awareness）和多状态等特性，以太坊智能合约与比特币脚本所能提供的智能合约相比，前者的功能更加强大。

6.3.2　以太坊账户

与比特币纯粹采用交易列表不同，在以太坊系统中，基本单元被称为“账户（每个账户为一个 20 字节的地址）”，以太坊网络追踪每个账户的状态，所有以太坊区块链的状态转换都是账户之间的信息和价值转移。

TIPS:

在以太坊中只有两类账户：①外部控制账户（Externally Owned Account，EOA），由一个私钥所控制；②合约账户，由它们的合约所控制，并且只能被外部控制账户“激活”。

对于大部分用户来说，这两类账户的区别是：EOA 账户通常是由人类用户来控制的，因为人类用户控制着私钥，进而掌控 EOA 账户；而合约账户都是由其内部代码管理的，如果说它被人类用户控制，那其实是因为通过编程使它受到 EOA 账户的控制，而 EOA 账户最终是由人类用户通过私钥来控制的。“智能合约”是指在合约账户中的代码，当交易发送到合约账户中时就执行该程序。根据以太坊的开发计划，到某个特定阶段，这两类账户的区别就排除了，所有的账户将统一为同一类型。

在 EOA 账户中有以太币余额，可以发起交易（以太币转让，或触发交易代码），它由私钥控制，但其中没有程序代码。在合约账户中也有以太币余额，还有程序代码，代码的执行由交易触发，或者由其他合约的信息调用触发，在执行代码时调用自己控制的永久内存，有自己的永久状态，并且可以调用其他合约。

以太坊区块链的所有动作都由 EOA 账户发起的交易来驱动，每当交易账户接收到一项交易时，由发送来的交易参数指引其代码的执行。合约代码由每个网络参与节点的以太坊虚拟机运行，这也是节点验证新区块的一个环节。

6.3.3 交易和消息

在以太坊中，“交易”是指一种采用加密签名的数据包，里面存储着从外部账户发往区块链上另一个账户的消息。交易的组成部分如下：

（1）消息接收方。

（2）验证发送者的身份的签名，这个签名同时说明发送方向接收方发送消息的意图。

（3）价值范围，用以说明从发送方转让给接收方多少“伟”的以太币。

（4）可选数据域，其中包含输送到合约的消息。

（5）启动燃料值（Start Gas Value），表示在交易执行过程中计算步骤允许使用的交易费用的最大数量。

（6）燃料价格（Gas Price）值，表示发送方愿意为燃料支付的费率（一个单位的燃料对应表示一条基本指令的执行，比如一个计算步骤）。

以太坊的“消息”则由一个合约发送给另一个合约，它是一种虚拟对象，不能序列化，只存在于以太坊的程序执行环境中，可以由函数调用产生。

消息包含如下组成部分：

（1）消息的发送方。

（2）消息的接收方。

（3）价值范围，是指与消息一起转移到合约地址的“伟”的数量。

（4）可选数据域，它是实际输入合约中的数据。

（5）启动燃料值，这一数据限定了由消息触发的代码执行所能引发变化的交易费用的最大数量。

TIPS:

消息与交易在数据结构上很相似，二者的根本区别在于：消息是由合约账户而不是由外部控制账户产生的，而交易是由外部控制账户产生的。当一个正在执行的合约执行到“调用（Call）”或“代理调用（Delegate Call）”代码时，就会产生和执行消息。

6.3.4　燃料

以太坊在区块链上的执行环境被称为“以太坊虚拟机”，参与网络的每个节点都要运行以太坊虚拟机，这也是区块验证协议的一部分。节点对所验证的区块中列出的交易进行验证，并运行以太坊虚拟机中的交易所触发的代码。网络中的每个节点都做相同的事，存储相同的值。

由于消息或交易的触发，在合约开始执行时，它的每条指令都在网络的每个节点上执行，这就产生了一种成本——每个执行的操作都要花费成本，采用燃料单位（Gas Unit）数来表示。

燃料（Gas）是指交易的发送方需要为在以太坊区块链上的每项操作支付的费用，包括计算、存储，这些费用要使用以太坊的内部加密货币——以太币来支付。为什么叫“燃料”这个名字？因为一种观点认为，交易费用的作用类似一种加密的燃油，可以驱动智能合约的运动。当以太坊虚拟机执行交易时，燃料将按照特定规则被逐渐消耗。燃料是从执行代码的矿工处购买的以太币，由于以太币的价格受到市场的影响处于波动之中，因此，在燃料单元与包含成本的计算单元匹配后，燃料与以太币是脱钩的。

TIPS:

> 燃料价格和以太币价格都是由市场自由调节的，以太币的价格根据市场情况波动，而燃料价格由矿工决定，如果燃料价格低于他们的最低要求，他们就会拒绝处理交易。燃料可以用用户账户中的以太币购买，以太坊客户端可以根据指定的交易最大支出限额，自动用以太币购买燃料。

这种交易费用的设计机制和比特币一样，使以太坊用户向网络支付少量交易费用，以防止恶意攻击和滥用以太坊网络，比如 DDoS 或无限循环攻击。每笔交易都要包含一个燃料限额，以及每个计算步骤支付的费率。根据交易提供的燃料额度，矿工可以选择打包交易并收取费用，或者拒绝处理交易。如果交易（包括可能触发的原始消息及任意子消息）产生的计算步骤所需消耗的总燃料量小于或等于用户愿意支付的燃料限额，则交易进入处理状态；反之，无论执行到什么位置，一旦所需的总燃料量大于燃料限额导致燃料被耗尽（比如降为负值），将会触发一个 out-of-gas 异常，那么该交易将全部取消，返回到原始

仅供非商业用途或交流学习使用

状态，但已收取的燃料费用不会退还，因为矿工已经完成了对应的验证工作。交易执行完毕后，未使用的燃料会以以太币的形式返还给发送者，用户不必担心被过度收费，因为矿工只收取所消耗的燃料费用。因此，发出具有高于估计值的燃料限额的交易是安全的。

在估算燃料费用时，要考虑两个影响因素：①gasUsed，是指交易所需消耗的总的燃料；②gasPrice，是指特定交易中用以太币计价的每单位燃料的价格。

$$总成本= gasUsed \times gasPrice$$

以太坊虚拟机中的每项操作都规定了一个消耗的燃料值，gasUsed 就是把所有执行的操作要消耗的燃料加总。表 6.2 提供了以太坊虚拟机对燃料收费的部分标准。

表 6.2　以太坊虚拟机对燃料收费的部分标准

操作名称	燃料费用（伟）	备　注
步（Step）	1	每个执行循环的默认费用
停止（Stop）	0	不收费
自行终止（Suicide）	0	不收费
哈希（SHA3）	20	
内存加载（sload）	20	从永久内存中取值
内存存储（sstore）	100	将值存储到永久内存中
余额（balance）	20	
创建（create）	100	创建合约
调用（call）	20	启动只读调用
内存（memory）	1	超出内存时按每个额外的字收费
交易数据（txdata）	5	交易的代码或数据，按字节收费
交易（transaction）	500	交易基本费用
合约创建（contract creation）	53 000	进入家园（Homestead）版后，燃料费用从 21 000 伟提高到 53 000 伟

一个用户构造了一项交易并签名，每个用户都可以指定他们所期望的燃料价格，甚至可以指定为 0。但是在 Frontier 版的以太坊客户端中，已经规定了默认的 gasPrice 是 0.05e12 伟，即 0.05×10^{12} 伟。

仅供非商业用途或交流学习使用

TIPS:

由于矿工工作就是为了获得交易费用收入，所以，如果所有的交易费用最低都是 0.05e12 伟，就很难让矿工接受费用为 0 的交易，这些交易就不能得到处理。

下面是一个燃料费用估算的例子。假设有一个合约，内容是把两个数据加（Add）到一起。在以太坊虚拟机中，Add 消耗 3 单位的燃料，使用默认的燃料价格估算的总成本是：

总成本=3×0.05e12=1.5e11（伟）

因为 1 个以太币等于 1e18 伟，所以上例中的交易总成本相当于 0.00000015 个以太币。

这是一个简化示例，因为它忽略了其他的费用，比如在相加之前要把两个数据传送到合约中的费用。

交易费用由验证网络的节点收取，这些节点就是“矿工”，他们接收、广播、验证和执行交易，然后将这些交易（包含对以太坊区块链上的账户的状态更新）打包到区块中，接下来矿工之间进行竞争，使自己的区块成为以太坊区块链的下一个区块，成功的矿工会收到挖矿的以太币奖励。创建一个区块的时间是 15 秒，奖励 5 个以太币。这一点与比特币的工作量证明（PoW）机制相似，因为目前以太坊的挖矿采用的还是 PoW 机制，尚未切换到 PoS 机制。

6.3.5　合约

在比特币网络中，交易其实只有一类，就是比特币的转账，所以其合约是一种单一的合约。但是以太坊是一种图灵完备的语言，而且使用了以太坊虚拟机设置，原则上可以实现各种各样的交易。要实现各类交易，就要通过合约来完成。

合约是代码（合约的函数）和数据（合约的状态）的集合，存放在以太坊区块链上的某个特定地址空间。合约账户可以向其他账户传递消息，也可以进行图灵完备计算。合约用一种以太坊特定的二进制格式（称为以太坊虚拟机字节代码）进行存储。



仅供非商业用途或交流学习使用

TIPS:

> 合约通常是用高级语言编写的，如 Solidity 语言，然后由 EVM 编译器编译为二进制代码，再上传到区块链。其他的高级语言还包括 Serpent 和 LLL 等。

Solidity 语言是一种与 JavaScript 相似的高级语言，允许用户开发合约并编译为 EVM 二进制代码。它是当前以太坊的旗舰语言，也是主流的语言。

Serpent 语言与 Python 语言相似，也可以用来开发合约并编译为 EVM 二进制代码。它最简洁、干净，融合了许多在编程模式上具有易用性的低级语言的特色，同时增加了专用于合约编程的特殊的域定义。Serpent 语言用 LLL 语言编译，不过目前 Serpent 语言已经过时了。

LLL 的全称是 Lisp Like Language，是一种类似于汇编语言的低级语言，它意味着非常简单，本质上它就是在以太坊虚拟机上直接进行编程的小型封装器。

1. 合约的作用

合约一般有 4 个作用。

（1）维护存储的数据。这些数据对其他合约有用，或者对外部世界有用。例如，有的合约模拟一种货币，有的合约存储了某个特定组织的会员名录。

（2）为具有复杂访问规则的 EOA 账户服务。这种合约通常被称为“转发合约”，其典型形式是：当给定的条件满足时，简要地转发输入信息给预定目标。例如，如果给定的条件是“3 个给定的私钥签名，至少取得 2 个后就转发”，那么一旦达到 2 个以上的私钥签名，则特定信息被转发。更复杂的转发合约会针对转发信息的不同特点设定转发条件，比如通过复杂的访问条件设置，满足后可以取消提款限额。

（3）管理执行中的合约，或者管理多用户之间的关系。例如，财务合约、一些特定中介机构的第三方托管，以及某些保险合约。这一类也包括一些开放式合约，即一方对任何其他方在任何时间保持开放。例如，一个合约是针对某一数学问题，向任何提交了有效解题方案的人支付奖金，或者向任何提供了算力资源的人支付费用等。



仅供非商业用途或交流学习使用

（4）给其他合约提供函数。这一类其实就是软件库。

怎样写一个合约？就像所有的编程语言用“Hello,World!”作为第一个例子一样，我们可以用 Solidity 语言编写一个合约，把“Hello,World!”放到区块链上。但是在 Solidity 语言中并没有直接进行字符串输出的函数，最接近的是使用一个 Log Event（日志事件）将字符串写入区块链中，语句如下：

```
contract HelloWorld {
       event Print(string out);
       function() { Print("Hello, World!"); }
}
```

在每次执行时，这个合约都会在区块链上创建一个日志入口，把参数“Hello,World!”打印到区块链上。

Solidity 合约的编译有多种方式可选。

（1）通过命令行使用 Solc 编译器。

（2）在由 geth 或 eth（安装时仍然需要 Solc 编译器）提供的 JavaScript 控制台中，使用 web3.eth.compile.solidity 方式编译。

（3）在线 Solidity 实时编译器。

（4）用于建立 Solidity 合约的 Meteor dapp Cosmo。

（5）Remix IDE：以太坊官方推荐的智能合约开发 IDE，易于使用，可以在浏览器中快速部署测试智能合约。

（6）以太坊钱包。

2. 合约的编译

例如，有一个简单的合约：

```
> source = "contract test { function multiply(uint a) returns(uint d)
{ return a * 7; } }"
```

这个合约提供了一个简单的乘法，调用一个正整数 a，然后返回 a*7。

在由 geth 提供的 JavaScript 控制台中，使用 web3.eth.compile.solidity 编译后的结果如下：

```
> contract = eth.compile.solidity(source).test
{
```



仅供非商业用途或交流学习使用

```
  code: '605280600c6000396000f3006000357c0100000000000000000000000000
0000000000000000000000000000000000000090048063c6888fa114602e57005b60376004
356041565b8060005260206000f35b600060078202905060 4d565b91905056',
  info: {
    language: 'Solidity',
    languageVersion: '0',
    compilerVersion: '0.9.13',
    abiDefinition: [{
      constant: false,
      inputs: [{
        name: 'a',
        type: 'uint256'
      } ],
      name: 'multiply',
      outputs: [{
        name: 'd',
        type: 'uint256'
      } ],
      type: 'function'
    } ],
    userDoc: {
      methods: {
      }
    },
    developerDoc: {
      methods: {
      }
    },
    source: 'contract test { function multiply(uint a) returns(uint d)
{ return a * 7; } }'
  }
}
```

从上述编译结果中可以看出，经过编译后的输出结果分为以下几个域。

（1）code 域：经过编译后的 EVM 二进制代码。

（2）info 域：编译器输出的附加的元数据，如合约语言、版本号、编译器版本号等。

（3）source 域：源代码。

（4）language 域：合约语言（如 Solidity、Serpent、LLL 等）。

（5）languageVersion 域：合约语言的版本号。



仅供非商业用途或交流学习使用

（6）compilerVersion 域：编译该合约的编译器的版本号。

（7）abiDefinition 域：应用程序二进制接口定义。

（8）userDoc 域：用户的“自然定义格式（Solidity 语言专用于以太坊的格式）”文本。

（9）developerDoc 域：开发人员的“自然定义格式”文本。

3. 合约的部署

接下来，我们用前面提到的数据，在区块链上创建并部署一个合约，用 EVM 代码发送一项交易到空白地址，代码如下：

```
var primaryAddress = eth.accounts[0]
var abi = [{ constant: false, inputs: { name: 'a', type: 'uint256' } ]
var MyContract = eth.contract(abi)
var contract = MyContract.new(arg1, arg2, ..., {from: primaryAddress,
data: evmByteCodeFromPreviousSection})
```

这个环节需要支付合约执行费用。如果发送账户（代码中 from 域中的内容）支付后，根据 EVM 的燃料消耗规则扣减相应的金额，然后该交易由矿工收集到区块中，达成共识后将区块写到区块链上，那么这项合约从此就永存在区块链上了。

合约中的数据，除上述的代码相关内容外，编译器还输出了一些数据，称为“元数据（Metadata）”，它们是合约的相关信息。当与一个不是自己创建的合约交互时，我们会希望保存它的相关信息或查看源代码。合约的作者一般都支持这种做法，他们通过在区块链上注册，或者使用第三方服务（如 EtherChain），使这些信息可以被看到。应用程序接口“admin”就提供了便利的方式获取任何合约选择注册了的数据包。具体代码如下：

```
// 取得合约地址信息，用于手动验证
var info = admin.getContractInfo(address) // 查找，取值，解码
var source = info.source;
var abiDef = info.abiDefinition
```

这项操作的底层逻辑是这样的：

首先，将合约信息上传到用 URL（统一资源标识符）定位的可被识别的某个地方，这个地方可以公开访问。



仅供非商业用途或交流学习使用

其次，任何人都可以找到这个由合约地址识别的 URL。

通过两步区块链注册就可以完成这些要求：第一步注册，在合约中调用“HashReg”，注册合约的哈希值和内容哈希值；第二步注册，在合约的“urlHint”中注册带有内容哈希值的 URL 地址。使用上述机制，可以充分地找到合约的地址及 URL，从而获取合约的元数据信息包。

所以，对于一个认真尽责的合约创建者来说，他应该遵循下列步骤创建合约：

（1）在区块链上布置合约。

（2）获得合约的 JSON 文件的合约信息。

（3）把合约信息的 JSON 文件部署到所选择的 URL 地址。

（4）按顺序注册代码哈希、内容哈希、URL 地址。

使用 JavaScript 应用程序接口（API）可以轻松地完成上述过程。调用 admin.register()函数可以从合约中解压出信息，在给定的文件中写出信息的 JSON 序列化数据，计算文件的内容哈希，最终把内容哈希注册到合约的代码哈希中。当把这一文件部署到任意 URL 地址后，就可以使用 admin.registerUrl()函数在区块链上注册 URL 地址和内容哈希值了。以下是示例代码：

```
source = "contract test { function multiply(uint a) returns(uint d)
{ return a * 7; } }"
// 用 Solc 编译
contract = eth.compile.solidity(source).test
// 创建合约对象
var MyContract = eth.contract(contract.info.abiDefinition)
// 从合约中提取信息，在给定文件中保存 JSON 序列号
contenthash = admin.saveInfo(contract.info, "~/dapps/shared/contracts/
test/info.json")
// 将合约发往区块链
MyContract.new({from: primaryAccount, data: contract.code}, function
(error, contract){
  if(!error && contract.address) {
    //计算内容哈希值，在 HashReg 中用代码哈希对其进行注册，使用地址发送交易，返回
内容哈希值以便于注册 URL 地址
    admin.register(primaryAccount, contract.address, contenthash)
    // 在此为 URL 地址部署 ~/dapps/shared/contracts/test/info.json 文件
    admin.registerUrl(primaryAccount, hash, url)
  }
});
```



仅供非商业用途或交流学习使用

6.3.6　智能合约示例

下面通过一个完整的简单智能合约的例子来详细说明以太坊的合约产生流程。这个例子是以太坊官方提供的智能交流互动合约，称为 Greeter，具体网址为 https://www.ethereum.org/greeter。这是一个用命令行方式创建的智能合约。如前所述，在以太坊上，智能合约是一种持有对象的账户，包含代码和函数，可以与其他合约进行信息交互，做出决定，存取数据，发送以太币给其他账户。智能合约由其开发者进行定义，但其执行过程、提供的功能范围由以太坊网络规定。而且如果某智能合约没有“自毁”设置，就会一直存在和运行下去。因此，一般的智能合约都会在最后编写“自毁”代码以释放资源。

这个 Greeter 智能合约依托以太坊区块链，它的功能是：任何人都可以向它输入内容，它依据得到的输入产生相应的互动。具体代码如下：

```
contract Mortal {
    /* 定义地址类型变量 owner */
    address owner;

    /* 该函数在初始化时执行，并设定合约所有者 */
    function Mortal() { owner = msg.sender; }

    /* 该函数给合约恢复资金 */
    function kill() { if (msg.sender == owner) selfdestruct(owner); }
}

contract Greeter is Mortal {
    /* 定义字符串型变量 greeting */
    string greeting;

    /* 合约执行时该函数也运行 */
    function Greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* 主函数 */
    function greet() constant returns (string) {
        return greeting;
    }
}
```



仅供非商业用途或交流学习使用

TIPS:

上述代码使用了 Solidity 语言，这是一种“可继承”的语言，其中有两个合约，一个是 Mortal 合约，另一个是 Greeter 合约，而且 Greeter 合约继承了 Mortal 合约的特点，从而使 Greeter 合约的编写更加简捷。

代码编写完成后，需要用编译器进行编译后才能部署到以太坊上。Solidity 语言的编译器有多种，命令行方式的有 Solc，图形界面的有 Remix，在此我们以命令行方式为例。

首先对上述代码进行编译，得到编译代码和应用程序二进制接口（Application Binary Interface，ABI），具体命令如下：

```
solc -o target --bin --abi Greeter.sol
```

该命令执行后，在调用目标的目录中产生两个文件：已编译代码和二进制接口文件，文件名分别为 Greeter.bin 和 Greeter.abi。接下来使用这两个文件进行合约的创建和部署，代码如下：

```
var greeterFactory = eth.contract(<contents of the file Greeter.abi>)
var greeterCompiled = "0x" + "<contents of the file Greeter.bin>"
```

至此，经过编译的代码就可以在 Geth 下使用了。现在需要对一些参数进行设定，比如问候语的具体内容。我们可以设定一句常用的问候语，如“Nice to meet you! What's your name?”，然后运行这一程序即可，代码如下：

```
var _greeting = "Nice to meet you! What's your name?"

var  greeter  =  greeterFactory.new(_greeting,{from:eth.accounts[0],
data:greeterCompiled,gas:47000000}, function(e, contract){
    if(e) {
      console.error(e); // 如果出错，则发出提示信息
      return;
    }

    if(!contract.address) {
      console.log("Contract transaction send: TransactionHash: " +
contract.transactionHash + " waiting to be mined...");

    } else {
      console.log("Contract mined! Address: " + contract.address);
```



仅供非商业用途或交流学习使用


```
      console.log(contract);
    }
})
```

这一程序运行时的情形是这样的：

在终端上输入

```
greeter.greet();
```

系统返回

```
'Nice to meet you! What's your name?'
```

程序执行完后，为了释放资源，还需要运行自毁功能，即：

```
greeter.kill.sendTransaction({from:eth.accounts[0]})
```

6.3.7　以太坊挖矿

以太坊目前仍在使用工作量证明机制进行挖矿，其哈希算法被称为 Ethash。开发团队曾经声称要转向权益证明机制，但一直未有推进。以太坊挖矿产生区块的时间间隔为 12 秒，每个区块产生后奖励 5 个以太币。

和比特币网络一样，在以太坊网络中，挖矿有 3 个方面的作用：挖出以太币；维护网络安全；验证计算。

在以太坊网络中，共识的达成也基于选择最高难度的区块，矿工产生区块，其他人检验区块的有效性。在其他的结构完整性标准下，比如比特币，区块的有效性由给定难度下的工作量证明机制来验证；但是对于以太坊而言，当它进入 Serenity 阶段时，它的共识机制将由工作量证明机制替换为权益证明机制。

这种工作量证明算法称为 Ethash（一种改进了的 Dagger-Hashimoto 算法），包括找到一个随机数输入算法中，使计算结果小于一个确定的难度门槛。为了找到一个这样的随机数，最优策略是不断枚举各种可能性（碰撞结果），直到发现这样一个随机数，这需要很高的算力；与此同时，验证找到的答案反而很容易而且廉价，有利于不同的网络节点通过验证结果成立从而快速达成共识。由于挖矿计算输出的结果有着统一的分布格式（因为它们都是一个哈希函数应用程序的计算结果），可以保证找到这样一个随机数所需的时间仅仅取

仅供非商业用途或交流学习使用

决于难度标准，因此可以通过动态调整难度值来控制创建新区块的时间间隔。

以太坊的区块链与比特币的区块链在不少方面很相似，但也有不同的地方，其中主要的差别在于区块链的架构：与比特币不同，以太坊区块中既包括交易列表的拷贝，也包括最近状态的拷贝（最近状态是指嵌入状态中的默克根哈希值的帕特里克树，更加精确）。除此之外，区块号和难度值也存储在区块中。

TIPS:

因此，从严格意义上来说，以太坊挖矿所用的哈希算法已经与比特币网络所用的哈希算法 SHA256 不完全相同了，而是 SHA256 算法的一种变体，一般称之为 Keccak-256 哈希算法。

以太坊协议规定，难度的动态调整目标是使全网创建新区块的时间间隔保持在 12 秒，这种“节拍”强化了系统状态的同步性，使恶意攻击者很难维持一个分叉（会导致“双花”现象）和修改历史数据，只有当攻击者掌控了超过全网络一半以上的挖矿能力（这就是所谓的 51%攻击）时，才能实现对合法区块的攻击。

和比特币相同，赢得区块创建竞争的成功矿工会收到奖励，但以太坊的奖励内容更加丰富。

（1）“赢区块”的静态奖励，包含 5 个以太币。

（2）区块内部所花费的燃料成本，这是一个以太币数量，取决于当前燃料价格。

（3）作为区块组成部分，包含“叔区块（Uncle）”的额外奖励，每包含 1 个 Uncle 区块可以得到额外的 1/32 个以太币的奖励。

TIPS:

在获胜矿工提交的区块中，全部交易的执行所消耗的燃料费用由每项交易的发送者支付。根据共识协议的规定，发生的燃料费用直接支付到矿工的账户。随着时间的推移，静态奖励将会减少。

Uncle 是过时的区块，因为以太坊的区块产生时间短，仅有 12 秒，所以会同时产生叔区块，它和父区块共同组成了当前区块的“祖先”（最多回溯 6 个区

仅供非商业用途或交流学习使用

块）。有效的 Uncle 区块会得到奖励，这是为了在分散挖矿奖励时中和网络延迟效应，因此增加了安全性［所谓的“幽灵协议（Ghost Protocol）”］。

Ethash 算法采用了 DAG 机制，它既包括由 Thaddeus Dryja 研发的 Hashimioto 算法，通过 I/O 限定的设计，即内存只需读取挖矿流程的少量参数，使得它具有“抗 ASIC 硬件挖矿”特性；还包括由 Vitalik Buterin 研发的 DAG 算法，这一算法使用了 Directed Acyclic Graphs（有向无循环图）的数据结构，可以同时实现耗内存的计算和省内存的验证过程。

在 Ethash 算法中，并不使用区块链作为数据源，而是采用一种定制的 1GB 数据集合，这一数据集合在每 30 000 个区块后，在区块的数据基础上产生一次更新，根据每个以太坊区块 12 秒的产生时间间隔，大约在第 4.2 天发生一次更新，这个时间窗口被称为“纪元（Epoch）”。

因此，在客户端挖矿时，可以采用预先生成或缓存的方式来存储 DAG 数据集，从而使轻节点（只有低档 CPU、不采用 ASIC 挖矿和小内存的节点）也可以进行挖矿，不像比特币网络，随着算力竞争的升级，最终只能由 ASIC 矿机挖矿，相应也减少了“中心化”的倾向。不过，“道高一尺，魔高一丈”，由于以太币的价格不断上涨，有些公司专门开发出了针对以太币的 ASIC 矿机。

DAG 数据集的产生代码如下（Python）：

```
NUM_BITS = 512

def encode_int(x):
    "Encode an integer x as a string of 64 characters using a big-endian
scheme"
    o = ''
    for _ in range(NUM_BITS / 8):
        o = chr(x % 256) + o
        x //= 256
    return o

def decode_int(s):
    "Unencode an integer x from a string using a big-endian scheme"
    x = 0
    for c in s:
        x *= 256
        x += ord(c)
```

仅供非商业用途或交流学习使用


```
    return x
```

执行代码如下：

```
from pyethereum import utils
def sha3(x):
    if isinstance(x, (int, long)):
        x = encode_int(x)
    return decode_int(utils.sha3(x))

def dbl_sha3(x):
    if isinstance(x, (int, long)):
        x = encode_int(x)
    return decode_int(utils.sha3(utils.sha3(x)))
```

其中，各个参数的定义如下：

```
SAFE_PRIME_512 = 2**512 - 38117      # 小于 2512 的最大安全素数

params = {
     "n": 4000055296 * 8 // 数位，数据库（4GB）的大小，要乘以 65536
     "n_inc": 65536,                 # n 值在每个纪元的增量，每年增长 882MB，
                                     # 纪元=20000，需乘以 65536
     "cache_size": 2500,             # 轻客户端的缓存大小
     "diff": 2**14,                  # 难度系数（在区块评估期间调整）
     "epochtime": 100000,            # 区块中一个纪元的长度
     "k": 1,                         # 父节点个数
     "w": w,                         # 用于模幂运算哈希
     "accesses": 200,                # 在哈希计算期间访问的数据集个数
     "P": SAFE_PRIME_512             # 哈希计算和产生随机数的安全素数
}
```

前面说过，DAG 只与区块高度有关，可以提前计算。如果在每个纪元转换时（每过 30 000 个区块）再来计算 DAG，那么第一个区块就会有一段等待的时间，造成系统的迟延。为了避免这种长时间等待，一般都会提前计算 DAG 值。geth 和 ethminer 命令都可以自动执行 DAG 生成，并在同一时间保留两个 DAG 值以平滑纪元的转换。

自动 DAG 生成可以由控制台在挖矿时启动和关闭，也可以由带着-mine 选项的 geth 命令默认启动。为一个任意纪元生成 DAG 的代码如下：

```
geth makedag <block number> <outputdir>
```

如为 360 000 纪元生成 DAG，就是“geth makedag 360000 ~/.ethash”。

仅供非商业用途或交流学习使用

TIPS:

需要注意的是，Ethash 算法在 Mac/Linux 环境下使用“~/.ethash”参数，在 Windows 环境下使用“~/AppData/Ethash”参数来生成 DAG，以便于它可以在不同的客户端环境及多重实例执行情况下共享。

综上所述，以太坊挖矿所用的 Ethash 算法基于：①一种大容量、瞬态性、随机产生的数据集所组成的 DAG 的证明；②尝试解决一种特定的限制（哈希碰撞，以找到特定难度的随机值）；③同时部分取决于一个区块头的哈希值。它的设计思路是：在低速 CPU 的环境下，哈希计算出快速的验证时间，在使用大容量、高带宽的内存时也能产生巨大的挖矿速度。这种大容量内存的需求意味着拥有大规模矿机的矿工所获得的优势有限，而高带宽内存的需求意味着在共享内存上堆积大量高速的处理器与单个处理器相比也只能获得有限的好处。这一点非常重要，它意味着对节点来说用矿池挖矿并无优势，从而减少了中心化的趋势。

6.3.8　以太坊区块

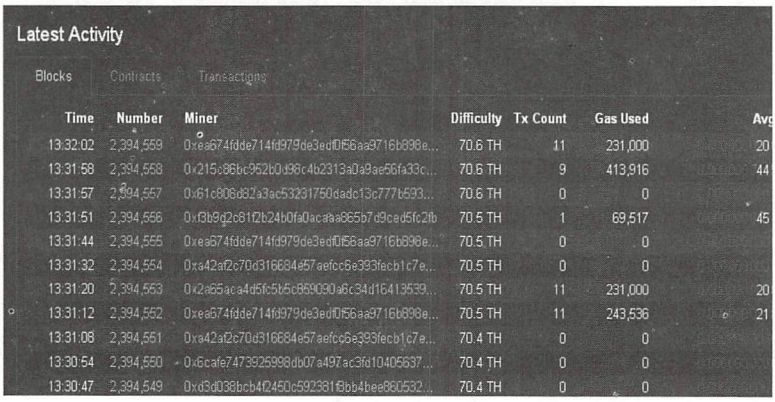
以太坊与比特币在很多方面比较相似，同时也都是在区块链技术基础上发展起来的，以太坊也有自己的数字货币——以太币，目前的挖矿算法也是 PoW，所以正如比特币一样，我们也可以借助软件对以太坊的区块进行浏览，看看其内部的数据构成。

1. 查看以太坊区块

要浏览以太坊的区块情况，可以借助第三方网站 https://tradeblock.com/ethereum/。进入该网站后，可以看到按创建顺序排列的区块列表，如图 6.2 所示。

在网页上可以看到每个区块创建的时间，以及区块号、创建区块的矿工地址、难度值、交易数量（Tx Count）、消耗的燃料量、平均燃料价格［单位：Gwei（吉伟）］。

仅供非商业用途或交流学习使用



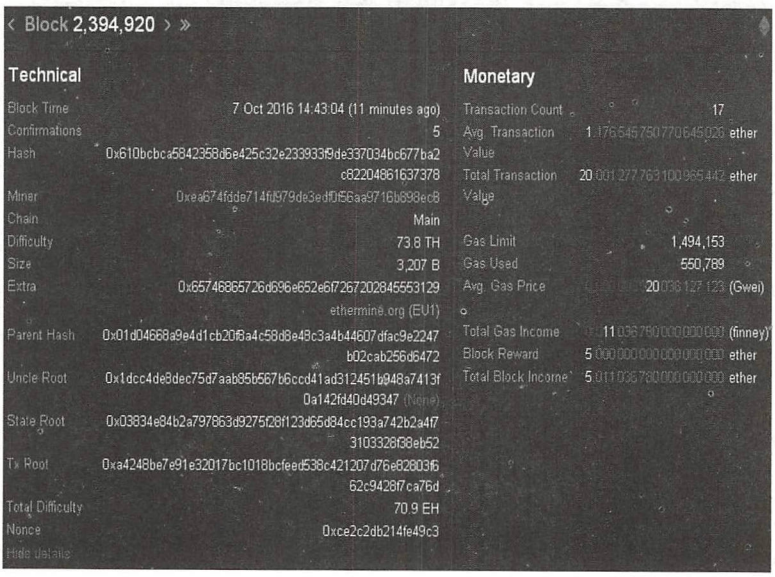
Latest Activity

Blocks Contracts Transactions

Time	Number	Miner	Difficulty	Tx Count	Gas Used	Avg
13:32:02	2,394,559	0xea674fdde714fd979de3edf0f56aa9716b898e...	70.6 TH	11	231,000	20
13:31:58	2,394,558	0x215c86bc952b0d86c4b2313a0a9ae56fa33c...	70.6 TH	9	413,916	44
13:31:57	2,394,557	0x61c806d32a3ac53231750dadc13c777b593...	70.6 TH	0	0	
13:31:51	2,394,556	0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb	70.5 TH	1	69,517	45
13:31:44	2,394,555	0xea674fdde714fd979de3edf0f56aa9716b898e...	70.5 TH	0	0	
13:31:32	2,394,554	0xa42af2c70d316684e57aefcc6e393fecb1c7e...	70.5 TH	0	0	
13:31:20	2,394,553	0x2a65aca4d5fc5b5c859090a6c34d16413539...	70.5 TH	11	231,000	20
13:31:12	2,394,552	0xea674fdde714fd979de3edf0f56aa9716b898e...	70.5 TH	11	243,536	21
13:31:08	2,394,551	0xa42af2c70d316684e57aefcc6e393fecb1c7e...	70.4 TH	0	0	
13:30:54	2,394,550	0x6cafe7473925998db07a497ac3fd10405637...	70.4 TH	0	0	
13:30:47	2,394,549	0xd3d038bcb4f2450c5923816bb4bee860532...	70.4 TH	0	0	

图 6.2　以太坊区块列表

单击任意区块的号码，可以查看该区块的详情。如图 6.3 所示，我们查看 2 394 920 号区块的详情。



< Block 2,394,920 > »

Technical

Block Time	7 Oct 2016 14:43:04 (11 minutes ago)
Confirmations	5
Hash	0x610bcbca5842358d6e425c32e233933f9de337034bc677ba2c822048861637378
Miner	0xea674fdde714fd979de3edf0f56aa9716b898ec8
Chain	Main
Difficulty	73.8 TH
Size	3,207 B
Extra	0x65746865726d696e652e6f7267202845553129 ethermine.org (EU1)
Parent Hash	0x01d04668a9e4d1cb20f8a4c58d8e48c3a4b44607dfac9e2247b02cab256d6472
Uncle Root	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347 (None)
State Root	0x03834e84b2a797863d9275f28f123d65d84cc193a742b2a4f73103326f38eb52
Tx Root	0xa4248be7e91e32017bc1018bcfeed538c421207d76e826036f662c9428f7ca76d
Total Difficulty	70.9 EH
Nonce	0xce2c2db214fe49c3

Hide Details

Monetary

Transaction Count	17
Avg. Transaction Value	1.176545750770645028 ether
Total Transaction Value	20.001277631009654 42 ether
Gas Limit	1,494,153
Gas Used	550,789
Avg. Gas Price	20.036 127 123 (Gwei)
Total Gas Income	11.036780000000000 (finney)
Block Reward	5.000000000000000000 ether
Total Block Income	5.011036780000000000 ether

图 6.3　以太坊 2 394 920 号区块的详情

网页上部显示的是区块的技术和资金两方面的情况。技术细节包括：

（1）区块创建时间。

（2）区块获得的确认数量。

（3）区块哈希值。

仅供非商业用途或交流学习使用

（4）创建区块的矿工的地址。

（5）区块链是否在主链上（如果在主链上，则显示为 Main）。

（6）难度值，单位为 TH。

（7）区块体积，单位为 B（字节）。

（8）外部区块地址。

（9）父区块哈希值，即上一个区块的哈希值。

（10）叔区块（Uncle）根，即上一个区块的兄弟区块的哈希值。

（11）状态根，即本区块的状态的哈希值。

（12）交易根，即本区块所有交易的默克根。

（13）总难度。

（14）随机数，以十六进制字符显示。

资金细节包括：

（1）区块中的交易数量。

（2）平均交易价值，用以太币计价。

（3）总的交易价值，即区块中所有交易的价值之和，用以太币计价。

（4）燃料限额。

（5）燃料消耗。

（6）平均燃料价格，单位为 Gwei。

（7）燃料总收入，即区块中各项交易支付的燃料之和，单位为芬尼。

（8）区块奖励，每个区块创建成功奖励 5 个以太币。

（9）区块总收入，用区块奖励加上燃料总收入，是矿工创建区块和验证交易的总收入。

在技术和资金情况下，网页上详细列出了本区块中所包含的交易列表，如图 6.4 所示。

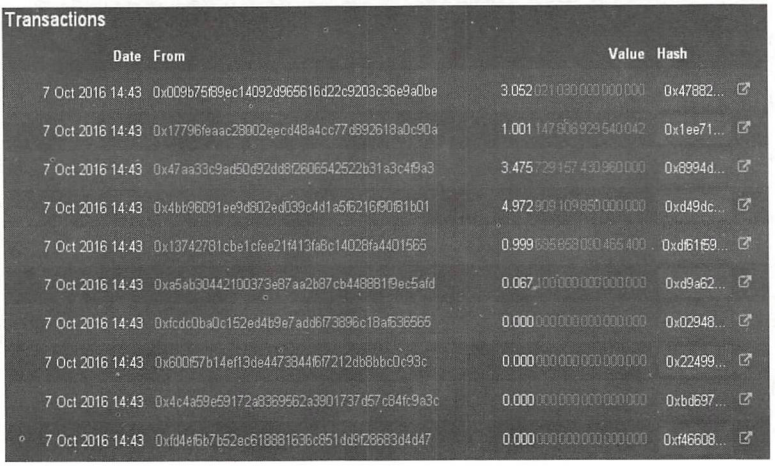


图 6.4　以太坊区块中的交易列表（部分）

在交易列表中，显示了每项交易的日期、来源（地址）、交易的价值金额、交易的哈希值，在最右侧是一个链接标记，单击后可以打开某项交易的详情。我们单击第一项交易的右侧链接，即可看到该项交易的详情，如图 6.5 所示。

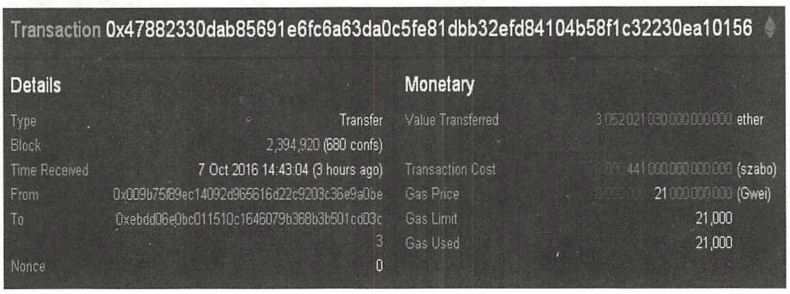


图 6.5　以太坊区块中的交易详情

从图 6.5 中可以了解到该项交易的类型为转让，包含在 2 394 920 号区块中，以及收到交易信息的时间、交易的发起方地址和目标地址。另外，还可以看到该项交易所涉及的资金情况，转让的价值为 3.052 021 030 个以太币，本项交易要花费 0.000 441 个以太币，燃料价格为 0.000 000 021 个以太币，燃料限额为 21 000，燃料消耗了 21 000。

还可以单击“From”下的账户地址，了解交易发起方的账户信息，如图 6.6 所示。

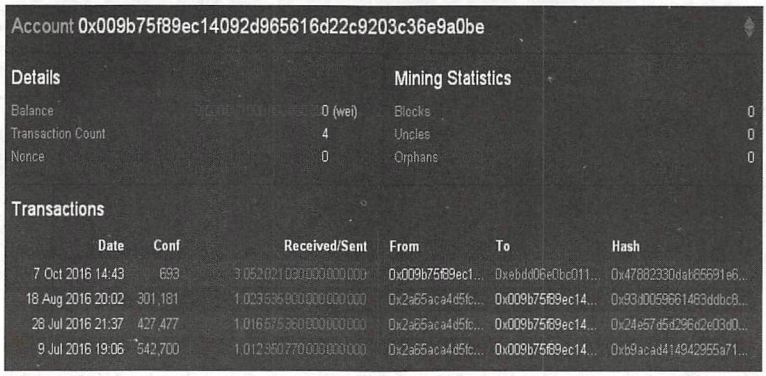


图 6.6　交易发起方的账户信息

从图 6.6 中可以了解到交易发起方（地址为 0x009b75f89ec14092d965616d22c9203c36e9a0be）的账户信息：该账户的以太币余额为 0，到目前为止共发生了 4 笔交易，在挖矿方面没有收获，创建的区块数、包含的 Uncle 区块数、孤区块数均为 0；下方还列出了与该账户相关的 4 笔交易的情况，有交易时间、获得的确认数量、接收或发送的以太币金额、每笔交易的来源和目标地址、每笔交易的哈希值等。

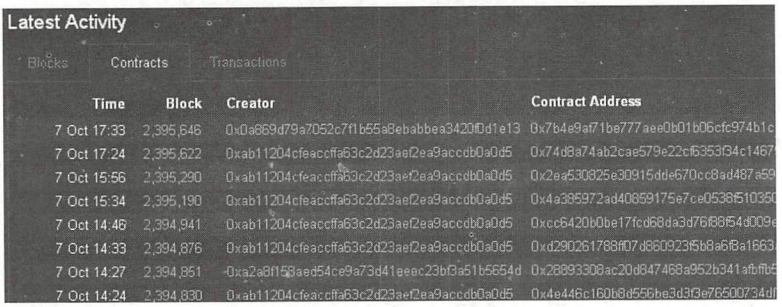
TIPS:

以上这些信息在以太坊区块中和在其他区块链应用的区块中相比差异不大，但是以太坊的特点在于采用“智能合约”作为核心功能，因此在区块中也包含了合约和交易信息。

2. 查看以太坊合约

在区块浏览器（https://tradeblock.com/ethereum/）上，除了可以看到区块的情况，还能看到合约和交易的情况。该网页上除 Blocks（区块）选项卡外，还有 Contracts（合约）和 Transactions（交易）两个选项卡，单击 Contracts 选项卡，就可以看到合约列表，如图 6.7 所示。

从图 6.7 中可以看到合约发出的时间、所属的区块号，以及创建者账户地址、合约地址。单击创建者账户地址，可以查看合约创建者的信息，如图 6.8 所示。

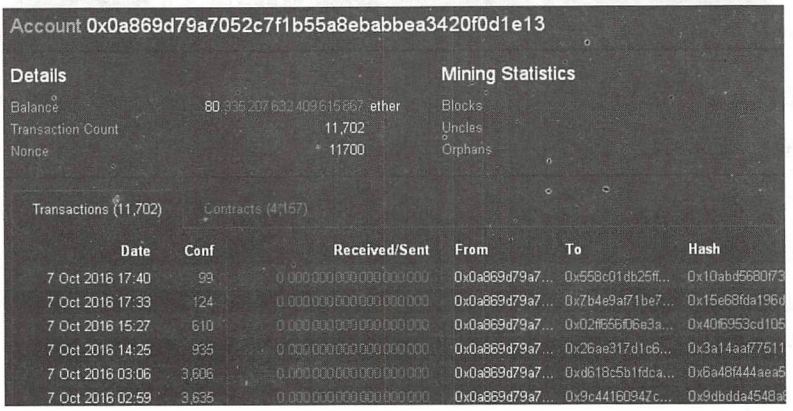


Latest Activity

Blocks　Contracts　Transactions

Time	Block	Creator	Contract Address
7 Oct 17:33	2,395,646	0x0a869d79a7052c7f1b55a8ebabbea3420f0d1e13	0x7b4e9af71be777aee0b01b06cfc974b1c
7 Oct 17:24	2,395,622	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0x74d8a74ab2cae579e22cf6353f34c1467
7 Oct 15:56	2,395,290	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0x2ea530825e30915dde670cc8ad487a59
7 Oct 15:34	2,395,190	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0x4a395972ad40859175e7ce0538f510350
7 Oct 14:46	2,394,941	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0xcc6420b0be17fcd68da3d76f88f54d009e
7 Oct 14:33	2,394,876	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0xd290261788ff07d860923f5b8a6f8a1663
7 Oct 14:27	2,394,851	0xa2a8f158aed54ce9a73d41eeec23bf3a51b5654d	0x28893308ac20d847468a952b341afbffb5
7 Oct 14:24	2,394,830	0xab11204cfeaccffa63c2d23aef2ea9accdb0a0d5	0x4e446c160b8d556be3d3f3e76500734df

图 6.7　以太坊区块中的合约列表



Account 0x0a869d79a7052c7f1b55a8ebabbea3420f0d1e13

Details

Balance　80.335 207 632 409 615 867 ether
Transaction Count　11,702
Nonce　11700

Mining Statistics

Blocks
Uncles
Orphans

Transactions (11,702)　Contracts (4,157)

Date	Conf	Received/Sent	From	To	Hash
7 Oct 2016 17:40	99	0.000000000000000000	0x0a869d79a7...	0x558c01db25ff...	0x10abd5680f73
7 Oct 2016 17:33	124	0.000000000000000000	0x0a869d79a7...	0x7b4e9af71be7...	0x15e68fda196d
7 Oct 2016 15:27	610	0.000000000000000000	0x0a869d79a7...	0x02ff656f06e3a...	0x40f6953cd105
7 Oct 2016 14:25	935	0.000000000000000000	0x0a869d79a7...	0x26ae317d1c6...	0x3a14aaf77511
7 Oct 2016 03:06	3,606	0.000000000000000000	0x0a869d79a7...	0xd618c5b1fdca...	0x6a48f444aea5
7 Oct 2016 02:59	3,635	0.000000000000000000	0x0a869d79a7...	0x9c4416094Zc...	0x9dbdda4548a8

图 6.8　合约创建者的信息

从图 6.8 中可以了解到，该项合约的创建者账户地址为 0x0a869d79a7052c7f1b55a8ebabbea3420f0d1e13，账户中以太币余额为 80.335 207 632 409 615 867 个以太币，共发生了 11 702 笔交易，随机数为 11 700，同时该账户在挖矿方面也没有收获。图 6.8 下方显示该账户发生的交易列表，不再详述。

如果单击合约列表中的“合约地址”，则可以查看某项合约的具体内容。比如，我们进入地址为 0x7b4e9af71be777aee0b01b06cfc974b1cdfb657b 的合约，可以看到如图 6.9 所示的内容。

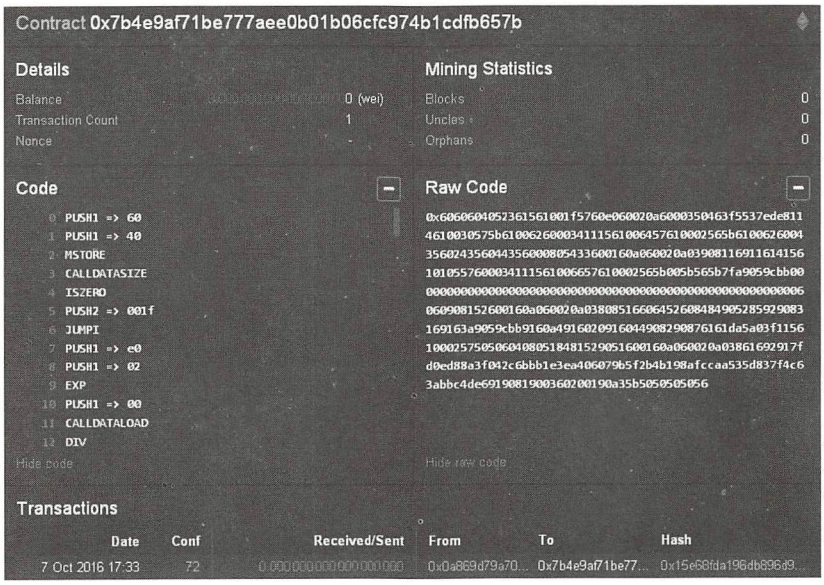


图 6.9　某项合约的具体内容

从图 6.9 中可以看出，该合约账户在挖矿方面无收获，账户中以太币余额为 0，发生的交易数量为 1 笔。该合约的代码也可以全部看到，内容如下：

```
0 PUSH1 => 60
1 PUSH1 => 40
2 MSTORE
3 CALLDATASIZE
4 ISZERO
5 PUSH2 => 001f
6 JUMPI
7 PUSH1 => e0
8 PUSH1 => 02
9 EXP
10 PUSH1 => 00
11 CALLDATALOAD
12 DIV
13 PUSH4 => f5537ede
14 DUP2
15 EQ
16 PUSH2 => 0030
17 JUMPI
18 JUMPDEST
19 PUSH2 => 0062
```

```
20 PUSH1 => 00
21 CALLVALUE
22 GT
23 ISZERO
24 PUSH2 => 0064
25 JUMPI
26 PUSH2 => 0002
27 JUMP
28 JUMPDEST
29 PUSH2 => 0062
30 PUSH1 => 04
31 CALLDATALOAD
32 PUSH1 => 24
33 CALLDATALOAD
34 PUSH1 => 44
35 CALLDATALOAD
36 PUSH1 => 00
37 DUP1
38 SLOAD
39 CALLER
40 PUSH1 => 01
41 PUSH1 => a0
42 PUSH1 => 02
43 EXP
44 SUB
45 SWAP1
46 DUP2
47 AND
48 SWAP2
49 AND
50 EQ
51 ISZERO
52 PUSH2 => 0105
53 JUMPI
54 PUSH1 => 00
55 CALLVALUE
56 GT
57 ISZERO
58 PUSH2 => 0066
59 JUMPI
60 PUSH2 => 0002
61 JUMP
62 JUMPDEST
```

```
63 STOP
64 JUMPDEST
65 JUMP
66 JUMPDEST
67  PUSH32  =>  a9059cbb0000000000000000000000000000000000000000000000000
000000000000
68 PUSH1 => 60
69 SWAP1
70 DUP2
71 MSTORE
72 PUSH1 => 01
73 PUSH1 => a0
74 PUSH1 => 02
75 EXP
76 SUB
77 DUP1
78 DUP6
79 AND
80 PUSH1 => 64
81 MSTORE
82 PUSH1 => 84
83 DUP5
84 SWAP1
85 MSTORE
86 DUP6
87 SWAP3
88 SWAP1
89 DUP4
90 AND
91 SWAP2
92 PUSH4 => a9059cbb
93 SWAP2
94 PUSH1 => a4
95 SWAP2
96 PUSH1 => 20
97 SWAP2
98 PUSH1 => 44
99 SWAP1
100 DUP3
101 SWAP1
102 DUP8
103 PUSH2 => 61da
104 GAS
```

```
105 SUB
106 CALL
107 ISZERO
108 PUSH2 => 0002
109 JUMPI
110 POP
111 POP
112 PUSH1 => 40
113 DUP1
114 MLOAD
115 DUP5
116 DUP2
117 MSTORE
118 SWAP1
119 MLOAD
120 PUSH1 => 01
121 PUSH1 => a0
122 PUSH1 => 02
123 EXP
124 SUB
125 DUP7
126 AND
127 SWAP3
128 SWAP2
129 PUSH32 => d0ed88a3f042c6bbb1e3ea406079b5f2b4b198afccaa535d837f4c
63abbc4de6
130 SWAP2
131 SWAP1
132 DUP2
133 SWAP1
134 SUB
135 PUSH1 => 20
136 ADD
137 SWAP1
138 LOG3
139 JUMPDEST
140 POP
141 POP
142 POP
143 POP
144 JUMP
```

上述代码经过编译后，原始代码显示如下：

```
0x60606040523615610010f5760e060020a6000350463f5537ede8114610030575b6
10062600034111561006457610002565b610062600435602435604435600080543
600160a060020a0390811691161415610105576000341115610066576100025 65b0
05b565b7fa9059cbb00000000000000000000000000000000000000000000000000000
000000606090815260016a060020a038085166064526084849052859290831691 6
3a9059cbb9160a491602091604490829087616 1da5a03f11561000257505060408 0
518481529051600160a060020a03861692917fd0ed88a3f042c6bbb1e3ea406079b
5f2b4b198afccaa535d837f4c63abbc4de691908190036020019 0a35b5050505056
```

3. 查看交易详情

在主页面上单击 Transactions 选项卡，即可看到最近发生的交易列表，如图 6.10 所示。

Latest Activity

Blocks　Contracts　Transactions

Time	Block	From	To	Hash	
18:33:03	2,395,843	0x2c7605890a1155496cf9...	0xbf4ed7b27f1d666546e3...	0xb9df87561d3308c96f7c...	0.000
18:33:03	2,395,843	0x1fa0e1dfa86b371fced62...	0x1e621321e99f3d632ad9...	0x04c8d0fa6058c86cc8cd...	0.000
18:33:03	2,395,843	0x6c7f03ddfdd8a37ca267...	0x705adcf67bda50ace76e...	0x1cad78cd936eb17e76a...	0.228
18:33:03	2,395,843	0xa027231f42c80ca4125b...	0x7ac003c0da6a77e419f2...	0x9fd2e6263a4feedb7fe11...	0.193
18:33:03	2,395,843	0x32be343b94f860124dc4...	0x92f08bbeb7a1ac1c37ac...	0x3d54e6203efc8bb5d2d1...	29.990
18:33:03	2,395,843	0x0e1d346bcc0456be420...	0x4aea7cf559f67cedcad0...	0x3f8e6dbd31e273fd8a22...	13.000
18:33:03	2,395,843	0x4c4a59e59172a836956...	0x37a9679c41e99db270b...	0x392a7b462115ba379ec...	0.000
18:33:03	2,395,843	0xfd4ef6b7b52ec6188816...	0x37a9679c41e99db270b...	0xf573683c3afc7f5fe0fc17...	0.000
18:33:03	2,395,843	0xfcdc0ba0c152ed4b9e7a...	0x37a9679c41e99db270b...	0xd98928f923e060774df0...	0.000
18:33:03	2,395,843	0x2c7605890a1155496cf9...	0xbb9bc244d798123fde78...	0xfbc0925adcb502cde69f...	0.000

图 6.10　最近发生的交易列表

从图 6.10 中可以看到收到交易信息的时间、包含在哪个区块中（区块号）、交易的来源和目标地址，以及交易的哈希值和交易涉及的金额。单击某项交易的哈希值，可以看到该项交易的详细情况，如图 6.11 所示。

从图 6.11 中可以看到，该项交易是一项合约调用（Contract Invocation）类型的交易，包含在 2 395 840 号区块中；还可以了解到区块接收到交易信息的时间、交易的来源和目标地址，随机数为 24 058，该项交易不涉及以太币的转让，交易费用为 0.000 827 307 287 538 740 个以太币，燃料价格为 0.000 000 022 792 090 130 个以太币，燃料限额为 235 000，实际使用了 36 298。

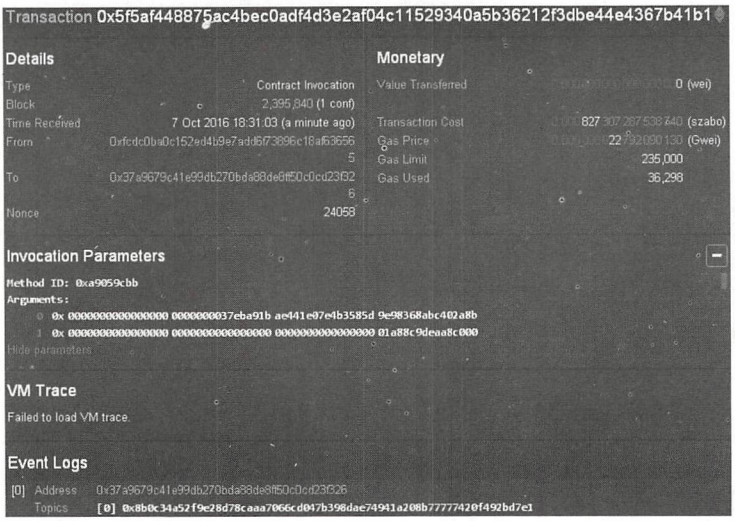


图 6.11　某项交易的详情

另外，从交易信息中还可以看到调用参数（Invocation Parameters）、追踪虚拟机的信息（VM Trace）、记录事件日志（Event Logs）。

综上所述，在以太坊的区块中，不但包含挖矿区块的情况，还包含相关的合约、交易的情况。对于以太坊网络的参与节点来说，在查阅区块信息并进行验证的过程中，信息更加丰富和透明，可以更加有效地完成验证；一旦验证完成，区块进入区块链，其安全性也比其他区块链应用有更大程度的提高。

TIPS:

更重要的是把合约和交易包含到区块链中，促进了“智能合约”的实现，增强了节点之间的信任程度，更好地满足了“去中心化”的要求。

chapter
SEVEN

第 7 章

以太坊应用开发基础

目前以太坊区块链是全球公认的“区块链 2.0”技术，相对于“区块链 1.0”的比特币网络来说，以太坊区块链采用以智能合约、数字资产为特点的应用模式，可根据不同的行业和人员进行定制化开发，为区块链技术在各行各业的应用和落地奠定了基础，为区块链技术走入社会生活打开了一扇全新的大门。在以太坊区块链出现和逐步成熟以后，区块链行业的发展逐步摆脱了仅局限于数字货币的狭窄范围，开始向金融、游戏、文化、权证等各方面发展，新的去中心化应用层出不穷，带动了区块链行业的极大繁荣。任何一个想要从事区块链相关工作的人，都不能忽视以太坊的开发技术。

7.1 以太坊开发环境的建立

以太坊的底层依然是分布式网络，其本身也是一种网络应用产品，当然，它是一种具备极强扩展性的网络应用产品。因此，要进入以太坊，需要通过客户端的形式与以太坊相连。当然，以太坊作为一种具备强烈的“去中心化”特点的网络，其客户端都能够实现以太坊网络的信息传送、验证功能，每个客户端之间是平等的，并没有“中心化”的超级节点存在。

TIPS:

以太坊有几种不同的客户端，分别采用不同的编程语言，如 C++、Golang、Python、Java 等，还有一种图形化开发客户端 Remix。其中比较常用的有 Geth（Golang Eth，Google 开发的一种编程语言，用于以太坊），以及 Solidity 语言，在以太坊开发中应用范围非常广泛。

通过节点上运行的客户端，我们可以和以太坊区块链进行同步和交互，创建钱包，发送或接收以太币。对于普通用户来说，如果不进行以太坊深度开发，仅利用以太坊进行以太币交易、编写智能合约，那么使用一款合适的钱包软件即可，它会整合不同的编程语言，如 Geth、Solidity 等。

节点交互界面的实现方式有两种：一种是 Geth 使用的 JavaScript 控制台，内含 web3.js 库用于节点之间的“交谈”；另一种是通过 JSON RPC 协议。JSON RPC 是一种状态无关的、轻量级的远程程序调用协议，其中，RPC 即 Remote Procedure Call 的缩写；JSON 是一种轻量级的数据格式，可以表征数字、字符串、序列值，以及以“元素名称/数值”成对出现的数据集合。在 JSON RPC 协议中定义了各种数据类型及其处理规则，并使用 JSON（RFC 4627）作为数据格式。

在初学以太坊应用开发时，一般采用测试网络，而不要轻易立即接入以太坊实时网络。因为如果接入以太坊实时网络，那么节点会自动与整个以太坊网络进行同步，下载当前所有的区块信息和交易信息，因此需要消耗较长的时间，还需要至少 20GB 的硬盘空间。

另外，根据以太坊的特点，当在实时网络上运行学习和测试用的智能合约时，以太坊需要支付以太币，这无疑增加了开发者的成本；而在本地的测试网络上运行则完全免费，也不用下载整个以太坊区块链，通过在本地建立以太坊网络的实例建立私人区块链，在应用开发过程中反应速度更快。因此，在以太坊应用开发过程中，有全节点模式与轻节点模式两种类型，分别采用不同的客户端。为了提高运行效率，我们选用轻节点模式。

TIPS:

运行以太坊客户端，需要下载 Mist 和 Mix 软件。Mist 是分布式应用（DApp）的浏览器和挖矿客户端，使得其他用户可以访问我们开发的分布

仅供非商业用途或交流学习使用

式应用，同时也是一个钱包软件，还可以在以太坊上部署智能合约。Mix 则是一个完全整合的开发环境，用来设计、建造和编译我们要开发的智能合约，编写和执行相应的前端代码。

如果希望使用效率更高的编程环境，那么采用 Geth 命令行模式是更好的选择。它具备 Mist 的全部功能，也是目前应用最广泛的以太坊客户端，因此从应用开发的适用范围来考虑，Geth 也是更好的选择。Geth 模式是一种全节点模式，将保持与全部以太坊网络的同步，对网络带宽、硬盘空间的要求较高，通常也要求实时在线。但是，作为一个入门级开发人员，在初期最好在 testrpc 上进行以太坊开发，因为以太坊应用的部署和在区块链上运行都需要付费（支付燃料），而通过 testrpc 方式在本机上模拟以太坊区块链，则不需要支付任何费用，这是上文所提到的轻节点模式。具体采用的开发软件是 Ganache CLI，它是 Ganache 的命令行版本，具有各种流行的远程程序调用函数，通过使用 ethereumjs 来模拟完全节点的行为，令以太坊应用开发更快捷、更容易、更安全。

采用 Web 方式运行的以太坊开发界面有 Remix，它是编程语言 Solidity 的集成开发环境，访问网址 https://remix.ethereum.org/即可进入其编程界面，可以在其中进行 DApp 的编程、编译和运行。

另一款用于轻节点模式的软件是 Parity，它由曾担任以太坊 CTO 的 Gavin Wood 所开发，采用 Web 界面，安装过程相对复杂，不过一旦安装完成，其性能比 Geth 有很大的提高。

此外，还可以选用 Truffle 客户端，它也是一种轻节点模式的客户端，可快速创建、编译、部署和测试区块链应用软件。

微软公司为了加入区块链的潮流，在其 Azure 云服务上提供了 ether.camp 这一成熟的以太坊开发环境。

读者可以根据自己的兴趣和条件自行选择相应的开发环境，在自己的电脑上进行安装。

仅供非商业用途或交流学习使用

7.2 Geth 开发环境

对 Geth 开发环境的介绍，我们以 Windows 操作系统为例。首先下载官方 Geth，即“摇摆舞 v1.8.12 版”，网址为 https://geth.ethereum.org/ downloads/，选择“Geth 1.8.12 for Windows”下载（注：由于该软件处于持续更新之中，读者可自行选择最新版下载）。下载后得到一个安装文件 geth-windows-amd64-1.8.12-37685930.exe，进行安装即可。

7.2.1 初次启动

安装后桌面上会出现 Geth 的快捷方式，双击运行，进入 Geth 命令行界面。因为是初次启动，所以它将与以太坊进行同步，下载所有的区块信息，包括区块头、数据、入口等，其中的明细包括计数、持续时间、区块序号、哈希值、区块大小等。根据网络带宽的不同，这一过程耗时较长。在国内通常的无线带宽情况下，以太坊区块已经超过 500 万个，要全部同步完成，需要持续数天时间。当然，Geth 支持“断点续传”，即在退出后重新进入时，可以从上次中断的区块开始接着进行同步。具体过程如图 7.1 和图 7.2 所示。

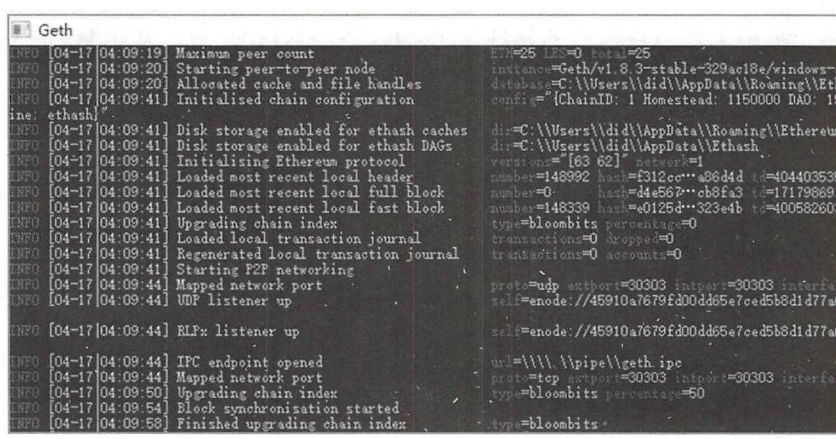


图 7.1 Geth 初始启动信息（部分）

仅供非商业用途或交流学习使用

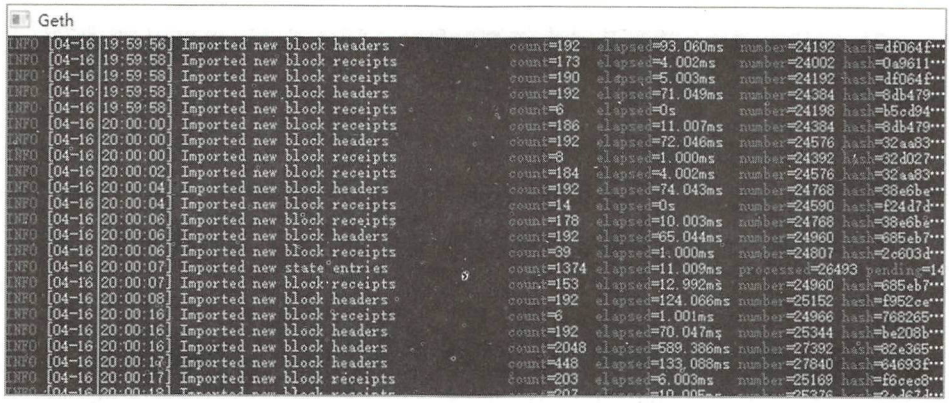


图 7.2　Geth 初次启动后同步以太坊区块的过程

如果需要加快同步过程，则可以在命令行模式下（Win+R 组合键）进入 Geth 的安装文件夹：

```
cd c:\users\AppData\Roaming\Ethereum
```

然后输入命令“geth -fast -cache=1024”，即可进行快速同步。

7.2.2　命令行安装模式

还有一种安装模式，即在 GitHub 的指导文件下进行 Windows 系统下的 Geth 安装，网址为 https://github.com/ethereum/go-ethereum/wiki/Installation-instructions-for-Windows。安装过程中不同的地方在于，当下载完 Geth 后，安装是在 Chocolatey 模式下进行的。

（1）Chocolatey 是一个强有力的 Windows 命令行模式平台，适用于对 Windows 下各种应用程序的各个方面的管理，包括安装、设置、更新及卸载，所以我们先进行 Chocolatey 的下载安装。在 https://chocolatey.org/网页中单击“Install Chocolatey now”，进入 Chocolatey 的安装页面，它要求在系统管理员的命令行模式（Power Shell）下输入命令来进行安装。我们可以用鼠标右键单击 Windows 10 的“开始”按钮，选择“windows powershell（管理员）”，进入命令行模式后输入如下命令：

```
Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.
sNet.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

仅供非商业用途或交流学习使用

（2）安装成功后，重新启动系统管理员命令行模式，可以采用 choco 方式来完成相关的程序安装任务。

输入“choco install git”命令，在系统中安装 Git 的 2.17 版本（注意：在安装过程中有一个提示“Do you want to run the script? [Y]es/[N]o/[P]rint:”，询问是否运行 chocolateyinstall.ps1 脚本文件，此时要选择输入 Y，以最终完成 Git 2.17 的安装）。

（3）输入“choco install golang”命令，在系统中安装 Golang 语言的 v1.10.2 Windows 64 位版本（Geth 是用 Go 语言写的，编译 Geth 源码需要 Go 语言和 C 语言编译器，因此需要先安装 Go 语言。Go 语言版本推荐用 Go 1.7 及以上）。

（4）输入“choco install mingw”命令，在系统中安装 MinGW v5.3.0 压缩包。

安装上述压缩包，是为了给 Geth 创造良好的运行环境和路径。

（5）进入常规命令行模式，输入以下命令，建立 Geth 的完整运行环境。

```
C:\Users\xxx> set "GOPATH=%USERPROFILE%"
C:\Users\xxx> set "Path=%USERPROFILE%\bin;%Path%"
C:\Users\xxx> setx GOPATH "%GOPATH%"
C:\Users\xxx> setx Path "%Path%"
C:\Users\xxx> mkdir src\github.com\ethereum
C:\Users\xxx> git clone https://github.com/ethereum/go-ethereum src\
github.com\ethereum\go-ethereum
C:\Users\xxx> cd src\github.com\ethereum\go-ethereum
C:\Users\xxx> go get -u -v golang.org/x/net/context
```

（6）最后导入全部的命令行参数，命令为“C:\Users\xxx\src\github.com\ethereum\go-ethereum> go install -v ./cmd/...”。至此就实现了在命令行模式下对 Geth 客户端的充分使用。

7.2.3 Geth 的使用

在 Geth 的具体使用方式上，又可以采取以下方式。

1. 基于以太坊主链的完全节点方式

人们使用以太坊网络的最常见场景是与以太坊主网络发生联系：创建账户、

仅供非商业用途或交流学习使用

传输资金、部署合约，并与合约进行数据交流和互动。在这种方式下，如果直接启动 Geth 命令，则开始与以太坊主链进行历史区块的同步。但实际上用户一般并不关心历史数据，只需同步以太坊最新状态即可。此时，在命令行模式下输入以下命令：

```
> geth console
```

即可实现对以太坊主链的存取和访问。

这条命令的功能包括：

（1）以快速同步的方式启动 Geth，通过与以太坊主链进行数据交换，既避免了大量下载历史数据，也避免了由此带来的 CPU 过载。

（2）启动了 Geth 内嵌的交互平台 JavaScript 控制台，由此可以调用所有官方的 Web3 函数和 Geth 自身的应用程序接口。

2. 基于以太坊测试网络的完全节点方式

这种方式主要是面向开发人员的。在创建真实的合约之前，人们会倾向于在开发、编程、测试过程中不产生真实的资金消耗，因为这样会大幅增加开发成本。但如果在编程和开发期间就采用上述第一种方式直接在以太坊主网络上进行工作，那么由于以太坊的 Gas 机制将会产生大量的 Gas 消耗，所以开发人员普遍倾向于在测试网络上开展工作。而在测试网络上，合约只使用虚拟的以太币，并不产生实际的资金消耗；只有在通过测试网络的验证、完全开发好后，合约才能被部署到以太坊主网络上，这就是基于以太坊测试网络的完全节点方式存在的意义。

具体的命令如下：

```
> geth -testnet console
```

其中，console 的含义是：在测试网络（testnet）上，具备了 Geth 的全部功能。

3. 基于 Rinkeby 测试网络的完全节点方式

上述第二种方式所用的测试网络是一种跨客户端的、基于 PoW（工作量证明）共识机制的测试网络，由于网络的安全性较低，易被攻击，同时会产生大

仅供非商业用途或交流学习使用

量的额外开销，因此以太坊社区的一些成员开发了 Rinkeby 测试网络，这是基于“授权证明（Proof-of-Authority）”的，这个网络更轻巧、更安全，但只支持 Geth。

具体的进入命令如下：

```
> geth -rinkeby console
```

7.2.4 Geth 在私有链上的应用

在编程使用 Geth 时为了加快速度，可以搭建私有链，在私有链上对智能合约进行测试和编译后再部署到以太坊主链上。下面通过一个例子来说明用 Geth 搭建私有链的过程。

1. 创建私有链

（1）首先新建一个文件夹，如 E:\EthPrivate，其作用是存储私有链相关数据。然后编辑一个 genesis.json 文件并存放在该文件夹下，这个文件是私有链的创始文件，其内容如下：

```
{
  "config": {
        "chainId": 43,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
  "alloc"      : {},
  "coinbase"   : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x20000",
  "extraData"  : "",
  "gasLimit"   : "0x2fefd8",
  "nonce"      : "0x0000000000000042",
  "mixhash"    : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp"  : "0x00"
}
```

仅供非商业用途或交流学习使用

TIPS:

> 注意，此处的“chainId”应该改为除 0 以外的任意随机数。如果 chainId 赋值为 0，则在最后的转账交易中会出现“insufficient funds for gas*price+value”报警，无法转账。但是当 chainId 设定为其他数值时，无此报警。

（2）在命令行模式下，进入 E:\EthPrivate 文件夹，执行以下命令，即可创建私有链。

```
geth --datadir data init genesis.json
```

运行结果如图 7.3 所示。

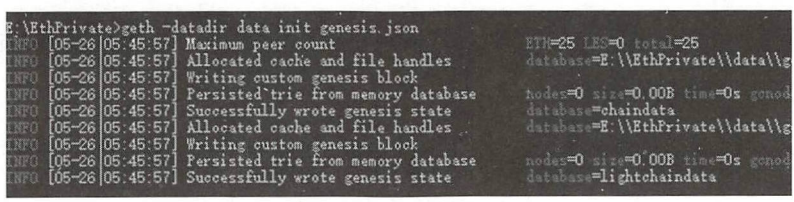


图 7.3　创建私有链的运行结果（部分）

（3）启动节点，连接这个测试用的私有链网络，输入以下命令：

```
geth --datadir data --networkid 123 --nodiscover console
```

其中，“networkid 123”的意思是测试网络的 id 命名为“123”，当然也可以命名为其他名称；“nodiscover”的意思是这是一个私有节点。运行结果如图 7.4 所示。

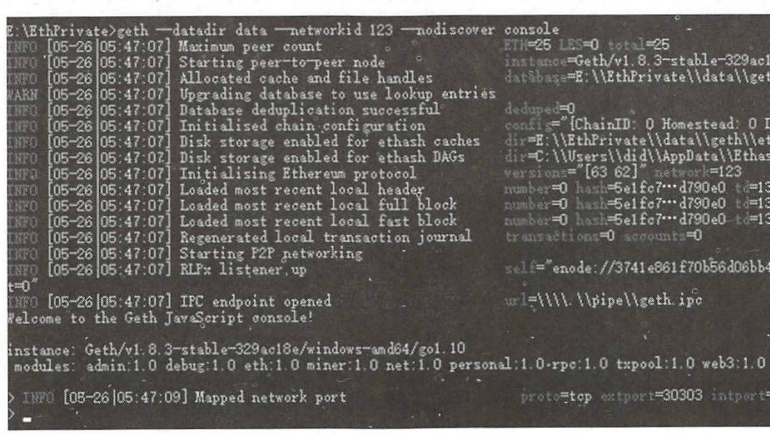


图 7.4　连接私有链网络的运行结果（部分）

仅供非商业用途或交流学习使用

这时可以看到，测试网络连接成功，在运行结果中有一句提示“Welcome to the Geth JavaScript console!”，表示我们已经进入 Geth 测试网络的交互式控制台模式。

2. 创建初始节点和账户

对于所有的节点来说，系统需要创建一个初始节点，它可以进行系统引导，作为参照系，让网络上的其他节点能够互相寻址和访问。这个初始节点的创建命令有以下两条：

```
bootnode --genkey=boot.key
bootnode --nodekey=boot.key
```

在创建了初始节点后，系统会显示一个 enode 网址，其他节点可以访问这个网址并发送点对点消息。

目前在这个测试网络中并没有账户，接下来我们先创建账户，过程如下。

（1）在命令行模式下进入 Geth 的交互式控制台模式，输入以下命令：

```
personal.newAccount("123456")
```

（2）这条命令将创建一个账户，其密码是 123456。运行结果是一个账户地址“0x655a2d67d74cb78807b08d6f3ae3c0e271eda3ee”，表明账户已经创建，此时可以在测试网络中查询到这个账户了。输入以下命令：

```
eth.accounts
```

系统显示“[0x655a2d67d74cb78807b08d6f3ae3c0e271eda3ee]”，表明账户已经存在。

（3）此时在这个账户中并没有余额。我们输入查询账户余额的命令：

```
eth.getBalance(eth.accounts[0])
```

运行后系统显示账户余额为 0。

（4）接下来开始在私有链上进行挖矿，为账户充值。

此时输入 3 条命令：

```
miner.start();admin.sleepBlocks(1);miner.stop()
```

其含义是：开始挖矿，当挖到第一个区块时即停止挖矿，将挖矿所得存入

仅供非商业用途或交流学习使用

第一个账户中。命令启动后，系统开始挖矿，经过一段较长的时间，系统不再显示“Generating DAG in progress”信息，表示已经挖矿成功，挖到的以太币存入了第一个账户中。

（5）这时再查询账户，输入以下命令：

```
eth.getBalance(eth.accounts[0])
```

系统显示“5000000000000000000”，这是用 wei 单位来表示的以太币数量，1 个以太币=10^{18}wei，因此这表示我们已经挖到了 5 个以太币。

3. 测试交易

接下来进行账户转账交易的测试，过程如下。

（1）创建另一个账户，输入以下命令：

```
personal.newAccount("456789")
```

运行后系统显示另一个账户名称“0xdca4e503bb4e5e8e4aea3ca0a0a6f36f53652767”。

（2）在交易之前，先解锁账户，输入以下命令：

```
personal.unlockAccount("0x655a2d67d74 cb78807b08d6f3ae3c0e271eda3ee",
“123456” ,3000)
```

这条命令的含义是：对第一个账户在 50 分钟（60 秒/分钟×50 分钟=3000 秒）内不需要输入密码即可解锁。运行后系统显示 true，表示解锁成功。

（3）输入以下命令：

```
eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1],value:
web3.toWei(3,"ether")})
```

这条命令的含义是：从第一个账户中转账 3 个以太币到第二个账户中。运行后系统显示如下结果：

```
INFO [05-27|21:22:56] Submitted transaction
fullhash=0xc903eb2701241f4f81120958f19ec56aa1ea937928438bfb0c5d8ce22b2d38a7
recipient=0xDCa4e503Bb4E5E8E4aea3Ca0A0a6f36F53652767
"0xc903eb2701241f4f81120958f19ec56aa1ea937928438bfb0c5d8ce22b2d38a7"
```

（4）此时由于系统没有启动矿工，交易还无人进行“打包到区块链中”的操作，所以再启动矿工，即输入命令“miner.start()”。因为这是测试私有链，交

仅供非商业用途或交流学习使用

易量很少，所以挖矿速度非常快，系统不断显示“Successfully sealed new block”，表示不断地挖矿成功。

（5）快速输入“miner.stop()”，系统显示 true，表示挖矿命令结束，转账交易也完成了。此时查询交易接收账户，可以发现已经有 3 个以太币（3000000000000000000wei）转到账户中了，而原账户由于矿工不断挖矿成功，余额也发生了变动，如图 7.5 所示。

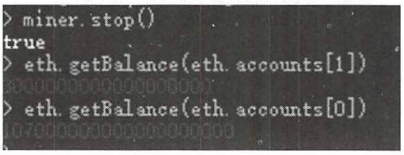


图 7.5　账户余额变动情况

（6）交易完成后，我们可以通过交易的哈希代码查询交易详情。输入以下命令：

```
eth.getTransaction('0xc903eb2701241f4f81120958f19ec56aa1ea937928438
bfb0c5d8ce22b2d38a7')
```

系统显示如图 7.6 所示的信息。

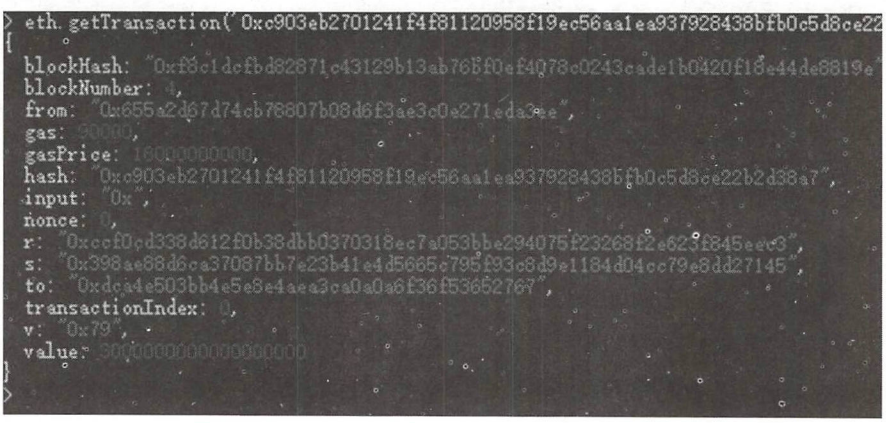


图 7.6　交易详情

以上通过一个测试私有链的例子展示了 Geth 客户端常用命令的用法。总的来说，当 Geth 客户端用于以太坊主链时，显得庞大而迟缓，但在测试网络的情况下效率很高。

7.3　轻节点模式

Geth 客户端与以太坊的同步是一个相当漫长的过程，也说明全节点模式对于初入门的开发者并不友好，所以我们建议初次接触以太坊应用开发从轻节点模式开始。可以采用 TestRPC 模式，即安装使用 Ganache GUI（Ganache 的客户端版本），其功能包括部署智能合约、开发去中心化应用及测试程序等。

Windows 系统的下载网址为“http://truffleframework.com/ganache/”，打开页面后单击右侧的“Download (Windows)”按钮，即可下载安装 Ganache GUI 版本，包括 Windows 版本、Mac 版本、Linux 版本，适用于不同的用户。

7.3.1　Ganache 图形界面

Ganache 的运行有两种模式：一种是 Windows 的图形界面模式；另一种是命令行模式。本书我们只介绍图形界面模式。安装完成后，运行 Ganache GUI，首次出现的主界面如图 7.7 所示。

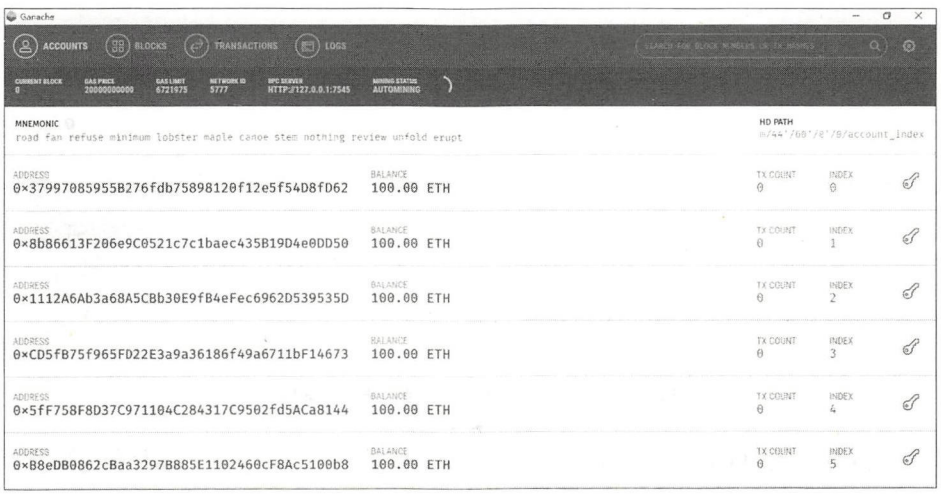


图 7.7　Ganache GUI 主界面（部分）

首次启动后，主界面显示服务器的详情，其中包括 10 个账户，每个账户已经预先分配了 100 个以太币，用于开发应用程序过程中需要测试时的 Gas 消耗，从而使用户能够专心从事自己的应用开发，无须操心通过挖矿来取得以太币。

在主界面上有 4 部分内容：一是账户（ACCOUNTS），其中显示了系统设定的初始账户信息和资产表，包括交易数量、索引值、私钥，为了保护私密信息，私钥是以钥匙图标显示的，只有单击后才弹出窗口显示该账户对应的私钥信息，每次进入 Ganache，都默认显示账户页面；二是区块（BLOCKS），将显示在区块链上挖矿所得的每个区块的信息，以及已消耗的 Gas 和交易；三是交易（TRANSACTIONS），这里列示了在区块链上运行的所有交易；四是登录（LOGS），显示的是每次打开 Ganache 时登录服务器的信息，常用于编译过程。

在 Ganache 主界面上单击鼠标右键后，在弹出的快捷菜单中选择“Inspect form”命令，出现编辑界面，其中显示了 Ganache 的各种要素的编辑信息（见图 7.8），将鼠标指针放置上去可以使图形界面中的对应要素的颜色发生变化。另外，也可以选择 Console（控制台）功能，即可进入命令行模式，可以输入编写的代码。

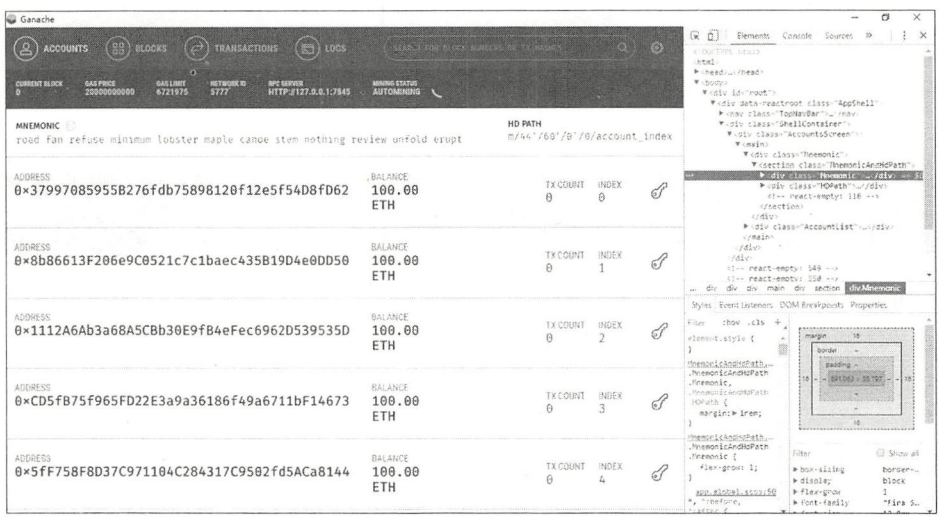


图 7.8　Ganache 的各种要素的编辑信息

可以看到，Ganache 初次启动后实际上并没有什么内容，因为它是以太坊开发的智能合约在私有链上的模拟部署软件，需要先进行智能合约的开发。为此我们需要安装以太坊的开发平台 Truffle，在 Truffle 上编写智能合约后，调用 Ganache 进行模拟测试，从而避免在真实的以太坊区块链上进行测试而导致的低效及 Gas 消耗。

7.3.2　安装 Truffle

在安装 Truffle 之前，需要先安装 Node.js，然后在 npm 模式下安装 Truffle。

1. 安装 Node.js

Node.js 是一个 JavaScript 运行时环境（Runtime Environment），发布于 2009 年 5 月，由 Ryan Dahl 开发，读者可以自行搜索查找，并下载 Windows 版本的 Node.js 进行安装。在安装过程中，除安装目录可以根据个人需求自选外，其他步骤都选择默认方式。安装完成后，可以先进行一下简单的测试，看看安装是否成功，以便开始进行环境配置。

（1）按 Win+R 组合键，输入“cmd”，然后按回车键，打开 cmd 窗口。

在命令行模式下输入“node -v”后按回车键，如果显示 Node.js 的版本号，则表示安装成功；否则还需重新安装。

npm 是随同 Node.js 一起安装的包管理工具，能解决 Node.js 代码部署方面的很多问题，常见的使用场景有以下几种：允许用户从 npm 服务器下载别人编写的第三方包到本地使用；允许用户从 npm 服务器下载并安装别人编写的命令行程序到本地使用；允许用户将自己编写的包或命令行程序上传到 npm 服务器供别人使用。

由于新版的 Node.js 已经集成了 npm，所以在安装了 Node.js 之后，npm 也同时安装好了。

TIPS:

在系统命令行模式下，可以通过输入“npm -v”来测试 npm 是否安装成功。命令运行后，如果出现版本提示则表示安装成功。同时，输入命令“npm install npm@latest -g”，npm 将自动进行更新。这是因为与 Node.js 较稳定的版本更新不同，npm 工具经常更新，我们需要保持 npm 为最新版本。

（2）进行环境配置。环境配置是对 npm 安装的全局模块所在的路径、缓存的路径进行重新设定，因为安装后在执行“npm install express -g”等类似的语句时，会将安装的模块安装到系统盘“C:\Users\用户名\AppData\ Roaming\npm”路径中，占用了 C 盘空间，且不便于维护，因此需要配置到自选的安装目录下。

在自选的Node.js安装文件夹下创建两个新的文件夹node_global及node_cache，然后在命令行模式下输入以下命令：

```
npm config set prefix "自选安装路径\nodejs\node_global"
npm config set cache "自选安装路径\nodejs\node_cache"
```

（3）关闭cmd窗口，进行环境变量的配置。按照“我的电脑”→单击鼠标右键→“属性”→“高级系统设置”→“高级”→“环境变量”的顺序，打开环境变量的设置窗口，在“系统变量”中新建一个“NODE_PATH”，输入“自选安装路径\nodejs\node_global\node_modules”，将“用户变量”下的“Path”修改为“自选安装路径\nodejs\node_global”，即可完成配置。

（4）配置完成后可以进行测试。例如，安装Node.js最常用的模块express，在命令行模式下输入“npm install express -g”，运行命令完成后显示“+ express@4.16.3 added 50 packages in 82.879s”，说明环境变量配置成功。

2. 安装 Truffle

执行完以上步骤后，终于完成了准备工作，接下来我们要安装Truffle了。

（1）按Win+R组合键输入cmd后，进入命令行模式，输入“npm install -g truffle”，系统将进行Truffle的安装，安装完成后将显示安装路径，以及Truffle的版本号，如“+ truffle@4.1.8”等信息。

TIPS:

> 还有一种方式测试Truffle是否安装成功，即在命令行模式下输入“truffle”，按回车键后如果显示truffle的主要命令，如init、compile等，则表示安装成功；或者输入“truffle version”，按回车键后显示版本号，也表示安装成功。

（2）安装成功后可以导入测试案例Metacoin。过程如下：

```
Mkdir Metacoin
cd Metacoin
truffle unbox metacoin（注：unbox命令，是Truffle专门用于下载已编写好的项目文档的命令）
```

输入后，系统会依次显示：

```
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!
Commands:
   Compile contracts: truffle compile
   Migrate contracts: truffle migrate
   Test contracts: truffle test
```

至此，说明 Metacoin 已经被导入 Truffle 中。因为是示例代码，所以在 test 中已经把测试程序写好了，可以直接运行。

7.3.3　运行 Ganache

（1）启动 Ganache，此时它是作为以太坊的客户端运行的。

（2）在命令行模式下（Win+R/cmd/）进入 Metacoin 文件夹下的 test 文件夹，运行 truffle metacoin.js 文件，这个文件即测试代码。

运行后屏幕显示：

```
using the network 'development'
```

这句的意思是：这是测试代码，使用的网络名为“开发”。后面的运行结果如图 7.9 所示。



图 7.9　测试代码在命令行模式下的运行结果

从图 7.9 中可以看到，Truffle 先后进行了 Deploying Migrations、Deploying

ConvertLib、Deploying Metacoin 等操作，最后是 Saving successful migration to network，即把迁移（Migration）的结果保存到“开发”网络上。

（3）经过上述操作后，用于开发的测试网络发生了变化，有一些交易被部署到了测试以太坊网络上，我们可以从 Ganache 界面上看到这些变化。在 Ganache 的默认启动界面上，可以看到，测试账户 1 的以太币余额减少了（因为部署交易消耗了 Gas），交易数量也不再为 0 了，如图 7.10 所示。

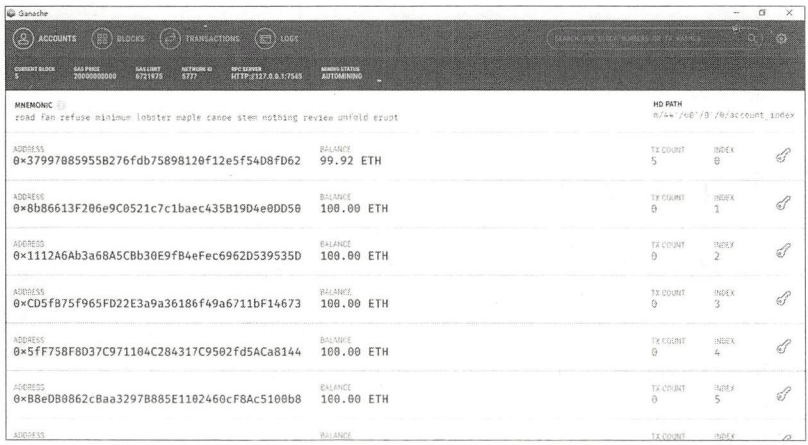


图 7.10　测试后的 Ganache 界面

（4）选择 TRANSACTIONS，查看交易详情，可以看到这 5 笔交易的信息如图 7.11 所示。

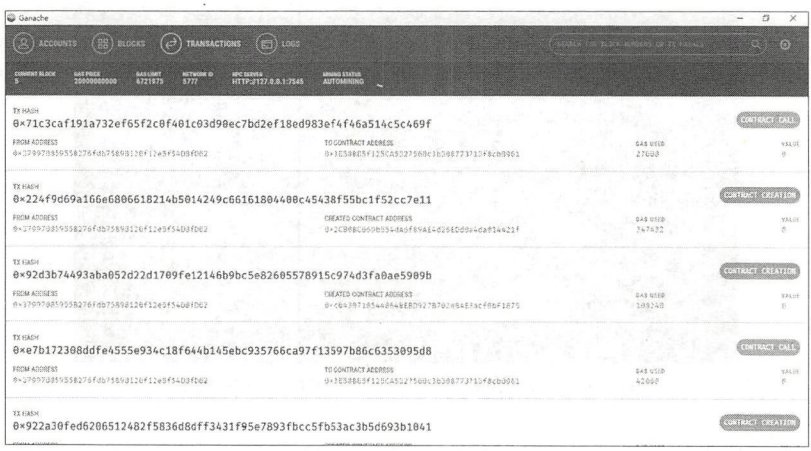


图 7.11　测试交易的详情

7.4　网页模式

针对以太坊 DApp 开发，当前最流行的编程语言是 Solidity，它是一种面向合约的编程语言类型，用来编写智能合约，属于高级语言的一种，运行在 EVM 上，其开发者有 Gavin Wood、Christian Reitwiessner、Alex Beregszaszi、Liana Husikyan、Yoichi Hirai 及以太坊前期核心团队的一些参与者，所以在以太坊中应用最为广泛，但实际上也可以用在其他各种区块链平台上。

TIPS:

前面讲述的两种模式都是命令行模式，虽然效率高，但是在 Windows 已经应用多年的情况下仍然要回到传统的命令行模式，令人感觉是一种历史的倒退，所以 Solidity 后来开发了基于浏览器的集成开发环境，这就是 Remix 模式，让开发者可以在浏览器模式下使用，这是一个很大的进步。

Remix 是一个开源的 Solidity 智能合约开发环境，提供基本的编译、部署至本地或测试网络、执行合约等功能。

在联网状态下，打开任何一款网络浏览器，输入网址“https://remix.ethereum.org/”，即可进入 Remix 的开发界面，如图 7.12 所示。

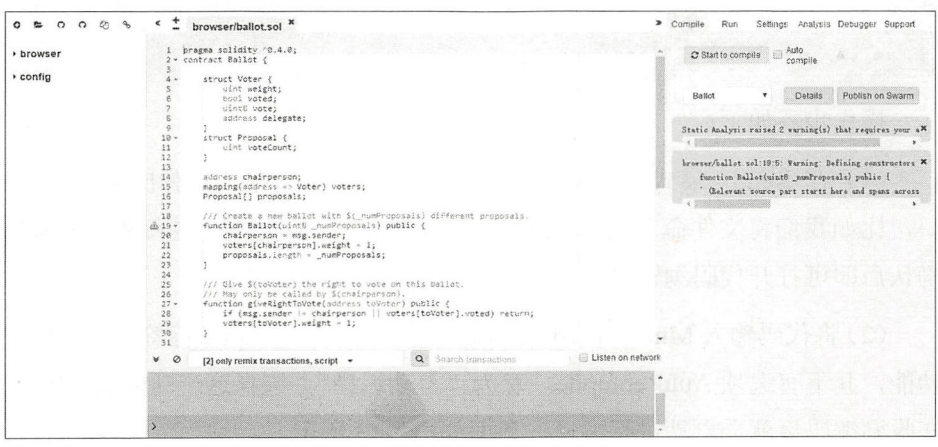


图 7.12　Remix 的开发界面

接下来我们通过以太坊官网的一个范例程序来了解 Remix 模式。在以太坊官网上有一个最小可执行的代币合约（Minimum Viable Token），其功能包括：

仅供非商业用途或交流学习使用

①创建代币，即在发起合约的以太坊账户中创建合约指定数量的代币；②代币转账，即向合约指定的以太坊账户转入指定数量的代币。

具体代码如下：

```
pragma solidity ^0.4.20;
contract MyToken {
/* 创建一个带有全部余额的数组 */
    mapping (address => uint256) public balanceOf;

    /* 用初始供应给合约创造器的代币来对合约进行初始化 */
    function myToken(uint256 initialSupply) public {
        balanceOf[msg.sender] = initialSupply;
        // 将所有的初始代币给予创建器
    }

    /* 发送代币 */
    function transfer(address _to, uint256 _value) public {
        require(balanceOf[msg.sender] >= _value);
        // 检验发送方是否已经持有足够的量
        require(balanceOf[_to] + _value >= balanceOf[_to]);
        // 过流检验
        balanceOf[msg.sender] -= _value;
        // 从发送者账户中减掉相应的代币
        balanceOf[_to] += _value;
        // 给接收者增加相同数量的代币
    }
}
```

操作步骤如下：

（1）在 Remix 模式下，最左侧有一个“+”图标，单击后弹出文件命名窗口，比如我们将文件命名为 MinimumToken.sol（注：文件扩展名必须为.sol），确认后即可打开代码浏览器，可以输入上述示例代码。

（2）将代码输入 MinimumToken.sol 后，在右侧有 Compile 标签，即“编译”功能，其下有选项 Auto compile，意为“自动编译”。选择这一功能后，Remix 在收到编码后就会自动进行编译，界面如图 7.13 所示。

仅供非商业用途或交流学习使用

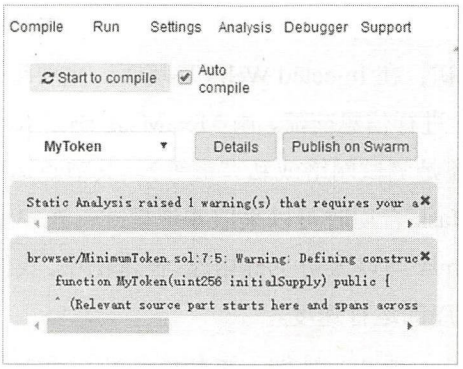


图 7.13　Remix 的编译界面

TIPS:

图 7.13 中显示了一条报警信息，即函数构造器中的函数名称 MyToken 与合约名称 MyToken 重名。为此将原来的代码中的函数名称 MyToken 改为 myToken，报警信息消失，编译正常。

（3）编译成功后，我们可以运行这一合约，选择右侧的 Run（运行）标签，此时的界面如图 7.14 所示。



图 7.14　Remix 的运行界面

（4）在图 7.14 中有 Enviroment（运行环境）选项，其中包括 3 个子选项：JavaScript VM、Injected Web3 和 Web3 Provider，分别代表不同的运行环境。

在 JavaScript VM 环境下，数据都保存在本地内存中，并不连接以太坊主网

仅供非商业用途或交流学习使用

络或其他测试网络、私有链等，只是一种在内存中存在的区块链，可以简单、快速地进行测试和挖矿；在 Injected Web3 环境下，主要用于与浏览器插件（如 MetaMask 钱包软件）进行信息交流，由 MetaMask 确定 Remix 与哪个网络相连（是以太坊主网络，还是测试网络或私有链等），其节点运行模式通常也不在本地，但是在与 MetaMask 合用时可以采取本地模式；在 Web3 Provider 环境下，可以通过 URL 将 Remix 与以太坊主链相连。在本例中，我们选择 JavaScript VM 环境，以提高测试程序的运行速度。

（5）运行 Account（账户）选项，系统已经默认建立了 5 个账户，并给每个账户中默认存入了 100 个以太币作为“启动资金”。我们选择第一个账户 0xca35b7d915458ef540ade6068dfe2f44e8fa733c（以其尾号 a733c 作为简称）。

（6）单击 Deploy（部署）按钮，它的功能是在内存测试网络中部署这个合约，这一动作会产生 Gas 消耗，从 a733c 账户（Account）中可以看到以太币的总量减少到 99.9999999999995833。另外，根据合约代码需要给定初始发币的数量，在界面上单击 MyToken at 0xbbf...732db(memory)，这是本合约的名称和地址，将会出现详细信息列表，如图 7.15 所示。

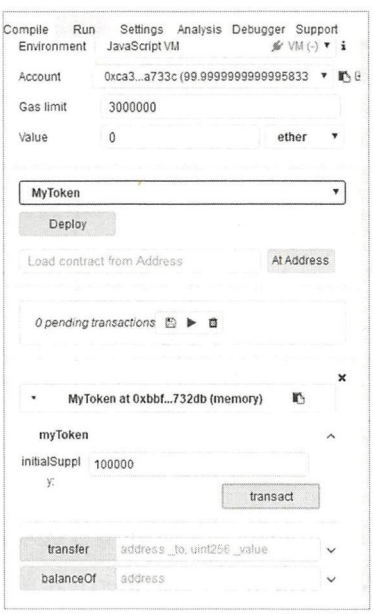


图 7.15　Remix 合约部署后的界面

仅供非商业用途或交流学习使用

（7）在 initialSupply 文本框中输入 100 000，意思是发行代币 100 000 个，单击 transact 按钮，系统将给初始账户创建 100 000 个代币（注意，这里的代币不是以太币，而是合约自己创建的代币，类似于当前市面上流行的各种数字虚拟货币），从“账户查询”中即可看到。即在图 7.15 中的 balanceOf 后面输入尾号为 a733c 的账户全部地址，然后单击 balanceOf，将在其下方显示“0:uint256:100000”，表示该账户中的代币余额为 100 000 个。

（8）接下来向第二个账户转入 1000 个代币，以此来测试“转账”行为。在图 7.15 中可以看到，transfer（转账）函数后面的参数为“address_to,uint256_value”，即先输入接收账户，再输入转出金额，然后单击 transfer 即可。为此我们输入系统创建的第二个账户的地址和金额——0x14723a09acff6d2a60dcdf7aa4aff308fddc160c 和 1000，单击 transfer，此时在界面下方的提示区显示转账成功，如图 7.16 所示。

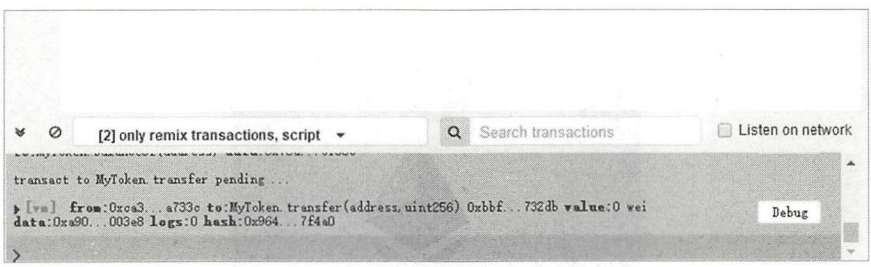


图 7.16　Remix 交易提示区信息

（9）最后使用 balanceOf 查询 0x14723a09acff6d2a60dcdf7aa4aff308 fddc160c 账户的代币数量，显示“0:uint256:1000”，即已经转入了 1000 个代币；同时也可以在 Account 账户中看到，尾号为 a733c 的账户的代币数量从 100 000 个减少到 99 000 个，表明代币转账成功。

7.5　Mist 浏览器

Mist 是以太坊官方提供的钱包软件和浏览器，同时也可以借助 Mist 连接开发者的私有网络，进行快速而高效的开发、测试和智能合约调试。因此，Mist 也可以作为以太坊应用开发的一个重要工具。除此之外，Mist 还是一款官方推

仅供非商业用途或交流学习使用

荐的钱包软件，任何人都可以通过 Mist 来存放自己的以太币，以及用于交易、买卖以太币等。

7.5.1 Mist 安装

下载 Mist 的官方网址是 https://github.com/ethereum/mist/releases，可以下载到 beta 版的 Mist 0.11.1 版（注：由于该软件处于持续更新之中，读者可自行选择最新版下载），我们选择 Win64 位版本的压缩文件进行下载。

下载完成后，像通常的软件一样对压缩文件解压后进行安装。

（1）找到 Ethereum Wallet.exe 文件，双击后即可开始安装 Mist 软件，电脑显示如图 7.17 所示的画面，表示本机系统正在与以太坊系统节点进行同步，根据网速不同，此处一般需要耐心等待。

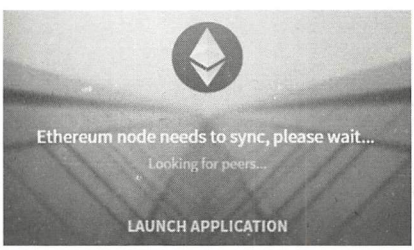


图 7.17　Mist 安装画面

（2）如果等不及了，则也可以单击下方的 LAUNCH APPLICATION（运行应用程序），Mist 将直接进入轻客户端界面，如图 7.18 所示。

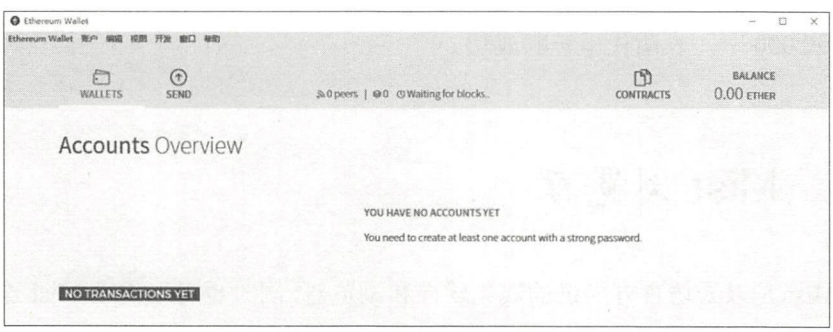


图 7.18　Mist 轻客户端界面

仅供非商业用途或交流学习使用

（3）单击菜单栏中的“开发”，其下显示“网络”子菜单，此时我们可以选择不连接主网（Ethereum），而连接测试网络，以便用于进行智能合约的开发和测试，一般选择 Ropsten-Test network，如图 7.19 所示。

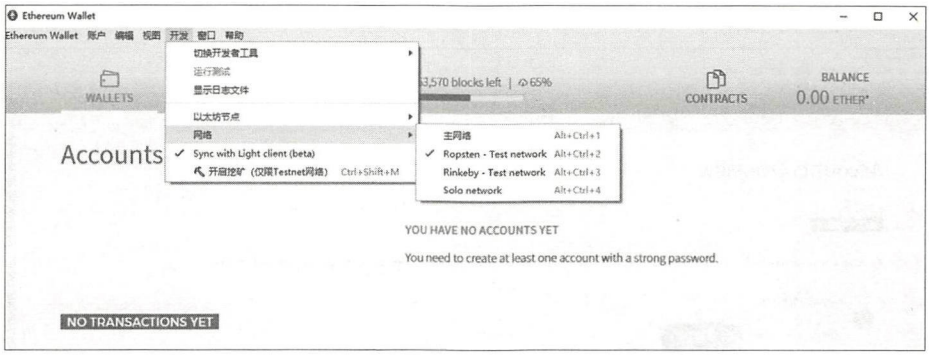


图 7.19　Mist 连接测试网络的选择

（4）选择连接测试网络之后，本机开始与测试网络进行区块同步，此时上方会有进度条和完成百分比的显示，如图 7.20 所示。

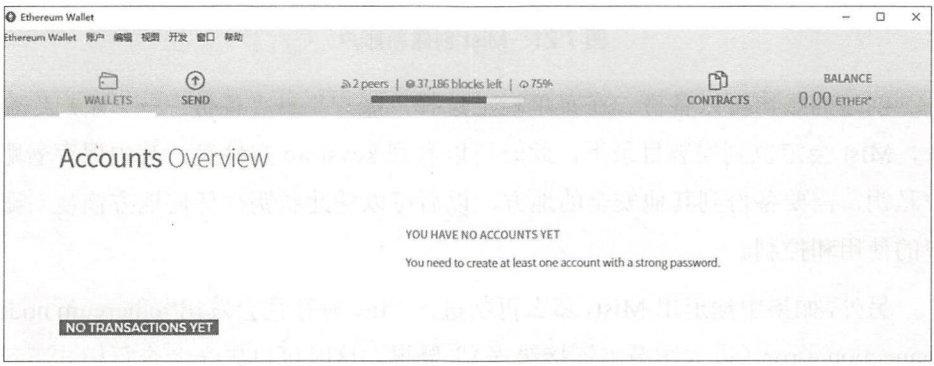


图 7.20　Mist 与测试网络进行区块同步

（5）现在我们再选择主网（Ethereum），在菜单栏的“开发”→“网络”命令下选择“主网络”即可，然后在主网下创建一个新账户，为以后的开发和测试做准备。

（6）在菜单栏上选择“账户”→“创建新账户”命令，Mist 弹出一个小窗口，要求输入密码。这个密码尽量设置得复杂一些，以提高账户的安全性，同时也要记住，因为在发送交易时需要使用密码。

仅供非商业用途或交流学习使用

输入一遍后，系统要求重复输入一遍，确认后按回车键，Mist 就自动创建出一个新的账户并显示出以 0x 开头的账户地址。当然，其中没有以太币，数量为 0.00，如图 7.21 所示（注：图中的账户地址信息尾部数据位由作者进行了马赛克处理）。

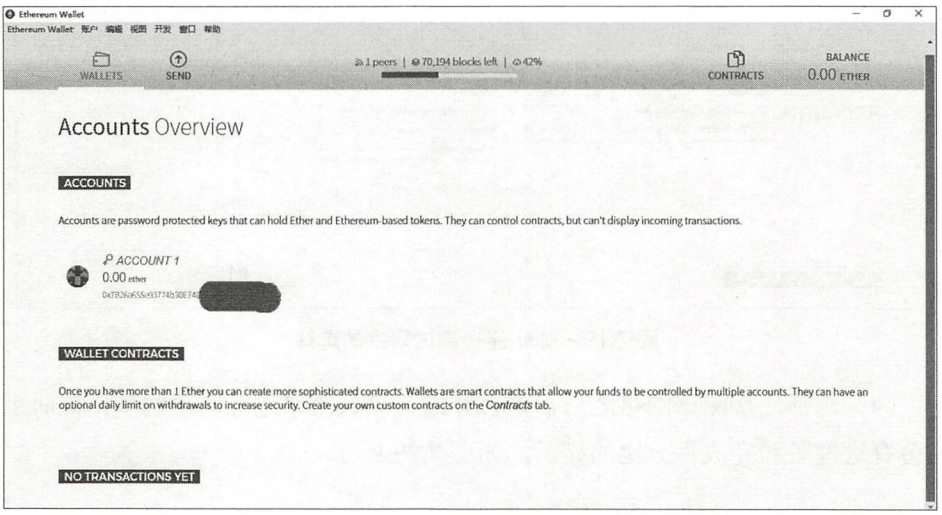


图 7.21　Mist 创建新账户

创建的账户可以备份。在菜单栏上选择“账户”→“备份”→“账户”命令，Mist 会定位到安装目录下，此时可以看到 keystore 文件夹，其中保存着账户私钥，需要备份到其他安全的地方，以后可以凭此私钥在任何地方恢复对账户的使用和控制。

另外，如果中途退出 Mist，那么再次进入 Mist 时往往会发出“ethereum node connection error（以太坊节点连接错误）”警报。这时可以先在命令行模式下运行 geth 命令，连接以太坊主网络，对区块进行一段时间的同步，再启动 Mist，就能连接上以太坊网络，不再出现“以太坊节点连接错误”警报。但与此同时会出现的是“Ethereum node connected”的“静止”画面，有可能几十分钟也不会变动，这一方面说明主网络连接正常，系统正在同步区块信息；另一方面说明同步的过程受到网速和服务器等的影响，需要很长时间，耐心等待后，系统还是会正常进入 Mist 的主画面的。

仅供非商业用途或交流学习使用

7.5.2　Mist 应用

本小节将过一个代币的创建合约部署过程来了解 Mist 的用法。网上有一份公开源代码的、用于学习和交流的代币创建合约，我们以此为例，了解 Mist 的具体应用过程。示例代码全文如下：

```
/*本代码在以太网上创建一种公共的、可交易的、可互换的代币，网址为
https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs

未编译的以下代码以“硬顶”创建加密资产，全部由合约创建者拥有。可创建函数改变这个合
约，比如允许担保、处置和冻结任意资产。本合约仅用于学习目的，使用者自行对其符合现在
和未来的金融、社交等数字世界的监管规则承担全部责任。
任何人都可以在任意目的（商业或非商业）下以任何方式免费复制、修改、出版、使用、编译、
销售或散发本软件，包括其源代码或编译后的二进制码。
根据版权法的司法管辖权，本软件的作者或作者们将本软件的所有版权捐赠给公众领域。我们
为了让这一捐赠对公众利益最大化而不惜损害我们的继承者的利益。我们同意这一捐赠行为公
开放弃了本软件在现有或未来版权法律下的所有权益。
本软件按“现状”提供，不提供任何形式的担保、明示或暗示，包括但不限于保证适销性、针
对特定用途的适用性和不侵权。在任何情况下，作者均不对来自与本软件或使用有关的或与之
相关的软件中的其他交易而导致的任何索赔、损害或其他义务承担责任，无论是在合同行为、
侵权行为还是其他方面。
要了解更多信息，请参考网址 http://unlicense.org */

pragma solidity ^0.4.21;
contract MyToken {
/* 代币的公共变量 */
string public name;
string public symbol;
uint8 public decimals;

/* 创建带有全部账户的数组 */
mapping (address => uint256) public balanceOf;

/* 在区块链上创建通知客户端的公共事件 */
event Transfer(address indexed from, address indexed to, uint256 value);

/* 用初始供给给合约创建者的代币数量对合约进行初始化 */
function MyToken(uint256 _supply, string _name, string _symbol, uint8
_decimals) public {
/* 如果供应不足，就创建 100 万个最小单位的代币 */
if (_supply == 0) _supply = 1000000;
```

仅供非商业用途或交流学习使用


```
/* 这些变量将保持不变，除非增加其他函数 */
balanceOf[msg.sender] = _supply;
name = _name;
symbol = _symbol;

/* 使代币可分拆，需增加小数数量的代币单位 */
decimals = _decimals;
}

/* 发送代币 */
function transfer(address _to, uint256 _value) public {
/* 如果发送方余额不足则停止发送 */
if (balanceOf[msg.sender] <_value) return;
if (balanceOf[_to] + _value < balanceOf[_to]) return;

/* 在新账户上进行增加或减少 */
balanceOf[msg.sender] -=_value;
balanceOf[_to] += _value;

emit Transfer(msg.sender, _to, _value);
}
}
```

（1）打开 Mist 后，单击 CONTRACTS 按钮，进入合约界面，上面显示了 3 个 “+”号按钮，分别是 DEPLOY NEW CONTRACT（部署新合约）、WATCH CONTRACT（观察合约）和 WATCH TOKEN（观察代币），如图 7.22 所示。

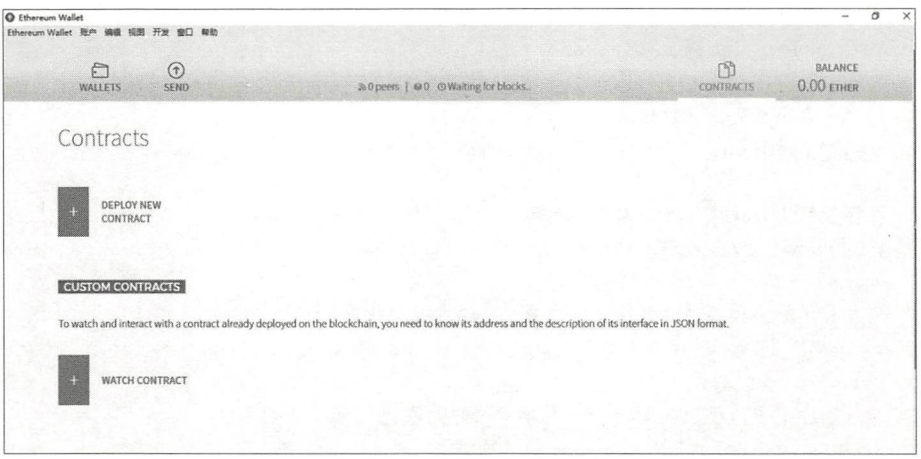


图 7.22　Mist 的合约界面

（2）在合约界面上单击 DEPLOY NEW CONTRACT 按钮，进入合约编辑界面，如图 7.23 所示。

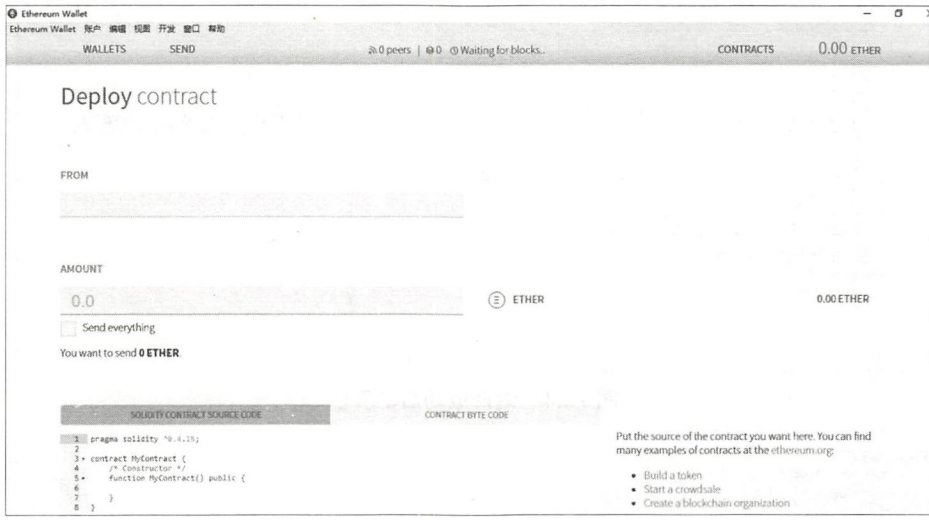


图 7.23　Mist 的合约编辑界面

（3）删掉原来左下侧 SOLIDITY CONTRACT SOURCE CODE（Solidity 合约源代码）区域内的代码内容，将上文所述的代码复制后粘贴进去，在右侧区域系统进行自动编译，修改程序 Bug。

TIPS:

对于开发者来说，这是一项特别友好的功能，因为其编译能力很强，错误提示准确、到位，有利于开发者编写出成功的智能合约。

（4）编译成功后，右侧出现 Pick a contract 提示，单击右侧的小三角，选择 My Token，如图 7.24 所示。

（5）接着会出现代币的各种设置参数，如发行总量（supply）、代币名称（name）、符号（symbol）、小数位数（decimals），并选择预计投入的费用（SELECT FEE），如图 7.25 所示。

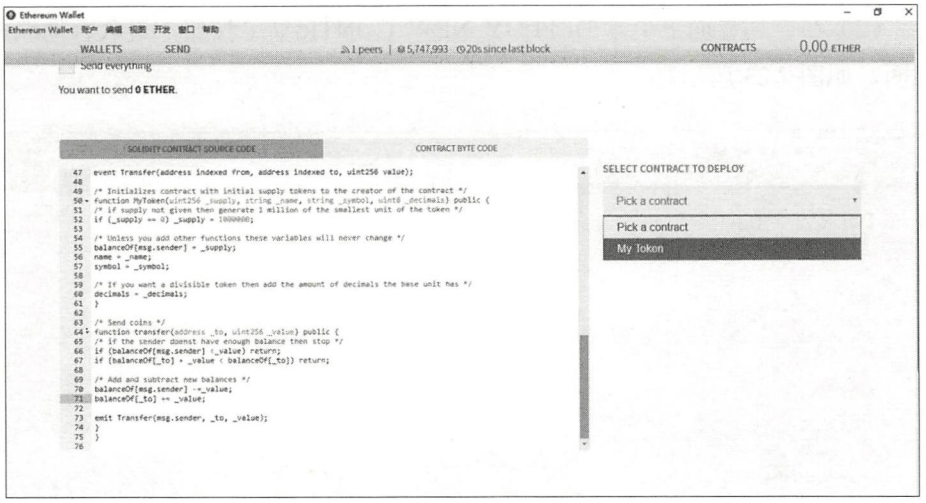


图 7.24　编译成功后的界面

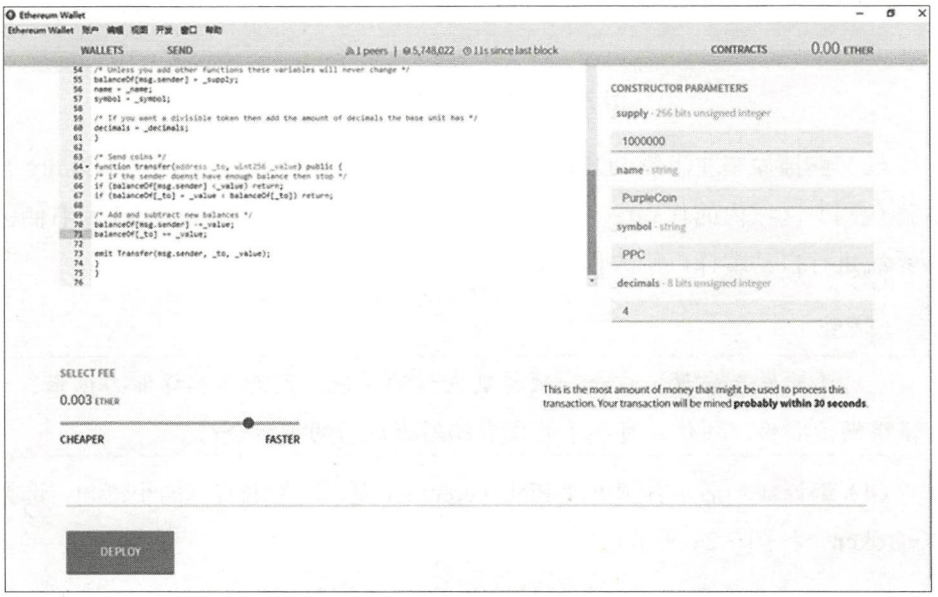


图 7.25　代币的参数设置

（6）单击左下侧的 DEPLOY 按钮，如果自己的账户中有购买的以太币用来支付发行费用，则在输入账户密码确认后，选择 SEND TRANSACTION（发送交易），系统回到 Mist 主界面，下方出现交易确认提示，经过 12 次确认后，合约部署成功。

（7）合约部署成功后，我们可以在 CONTRACTS 中看到自己创建的代币，并且可以完成向其他账户发送交易、转账等功能，其操作均非常直观，不再详述。

TIPS:

需要提醒大家的是，每项操作都需要支付相应的费用，所以开发者自己的账户里需要有一定的以太币存量，否则操作执行不了。

至此，我们应用 Mist 软件进行的代币开发就实现了其基本功能。

7.6　用 MetaMask 建立开发账户

MetaMask 是以太坊上比较流行的一种轻钱包（Wallet）软件，它最大的特点是“轻”，因为它不需要像其他软件一样安装使用，而是作为谷歌浏览器 Chrome 的一个插件，采用添加插件的形式即可使用。

既然是谷歌浏览器 Chrome 的一个插件，那么首先需要安装 Chrome 浏览器，此处不再详述；然后在其“应用商店”中找到 MetaMask，或者打开网址 https://metamask.io/（这是 MetaMask 的官方下载网站），也可以找到 MetaMask（注：经测试，MetaMask 作为插件也可以应用于搜狗浏览器）。具体步骤如下。

（1）从官方网站上进入 MetaMask。首先看到的是一幅抽象派风格的画面：一只狐狸头像出现在画面中间，它还可以随着桌面上鼠标指针的移动不断地改变自己的朝向去追踪鼠标指针，显得生动活泼，如图 7.26 所示。

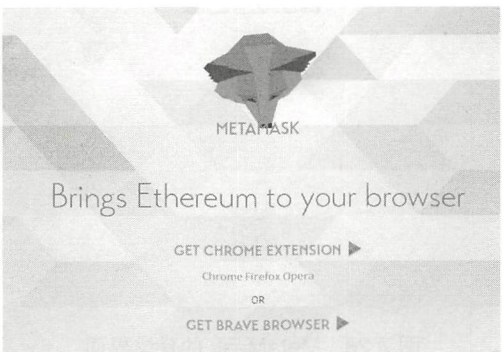


图 7.26　MetaMask 的进入界面

仅供非商业用途或交流学习使用

（2）单击 GET CHROME EXTENSTON 即可进入 Chrome 的应用商店，会自动弹出 MetaMask 的介绍页面，单击“添加至 CHROME”按钮即可下载这个插件到 Chrome 中，如图 7.27 所示。

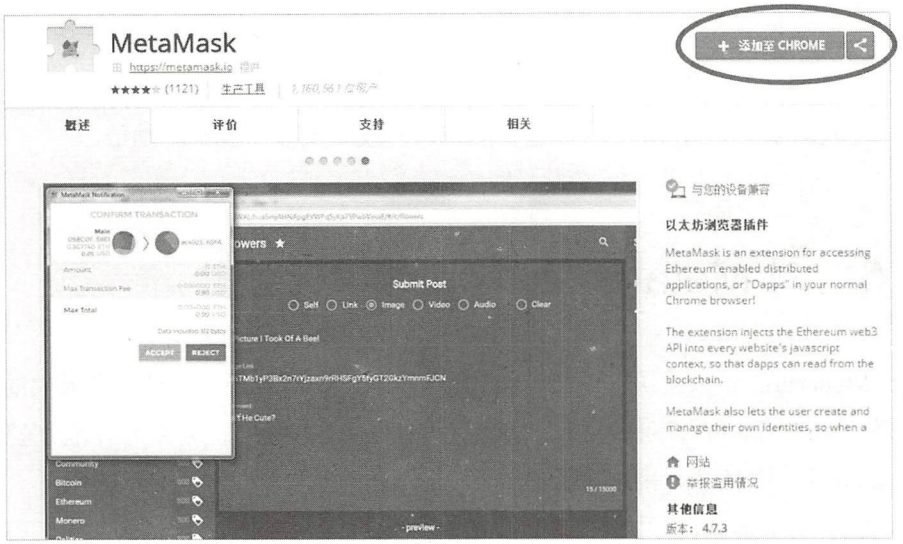


图 7.27　下载 MetaMask

（3）MetaMask 插件安装完成后，在 Chrome 浏览器右上角就会出现 MetaMask 的小狐狸图标，单击后即可出现登录界面，如图 7.28 所示。



图 7.28　MetaMask 的登录界面

仅供非商业用途或交流学习使用

（4）初次登录需要创建账户。输入至少 8 位数的密码，确认后单击 CREATE（创建）按钮，系统将会创建一个新账户；如有已经取得的以太坊账户，也可以单击 Import Existing DEN 按钮来导入原账户，如图 7.29 所示。



图 7.29　MetaMask 创建账户

对于创建完成的账户，系统会给出账户地址，并生成一段由随机顺序的 12 个英文单词组成的种子词汇（Seed Words），用于遗忘密码时的恢复。这是相当重要的信息，甚至比密码更加重要，需妥善保存。

（5）完成了账户的创建，即进入 MetaMask 的主界面，此时可以选择网络连接类型，比如连接主网络（Main Ethereum Network），或者连接各种测试网络，通过单击右上角的选择框进行选择。在连接主网络时，如果部署合约则需要消耗以太币作为 Gas，所以我们在编程和开发阶段一般选择 Ropsten Test Network，如图 7.30 所示。

（6）此时可以看到账户中以太币数量为 0，但在测试网络 Ropsten Test Network 中，可以申请测试用的免费以太币，具体操作为：先单击 DEPOSIT（存款）按钮，进入 MetaMask Ether Faucet 页面，这是发放免费以太币（测试用）的页面，再单击 request 1 ether from faucet 按钮，可以从 Faucet 这里获得 1 个以太币，即可用于支付测试智能合约部署的 Gas 费用，如图 7.31 所示。

仅供非商业用途或交流学习使用

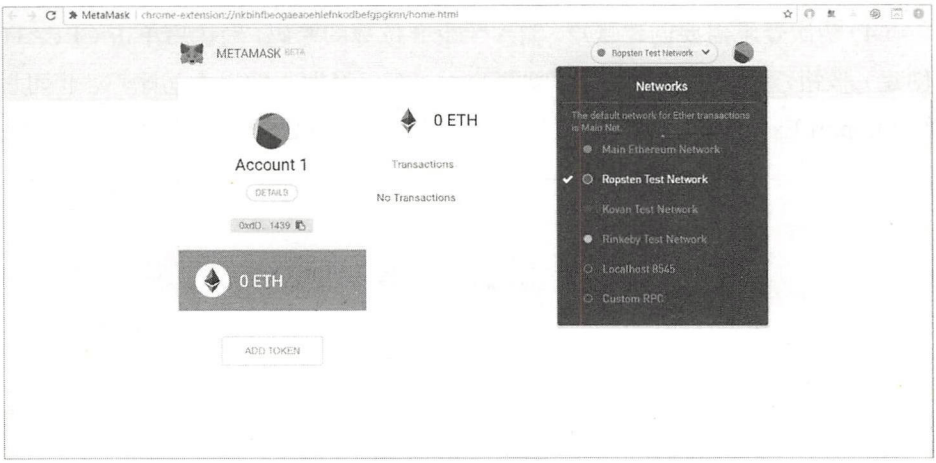


图 7.30　MetaMask 的连接网络选项

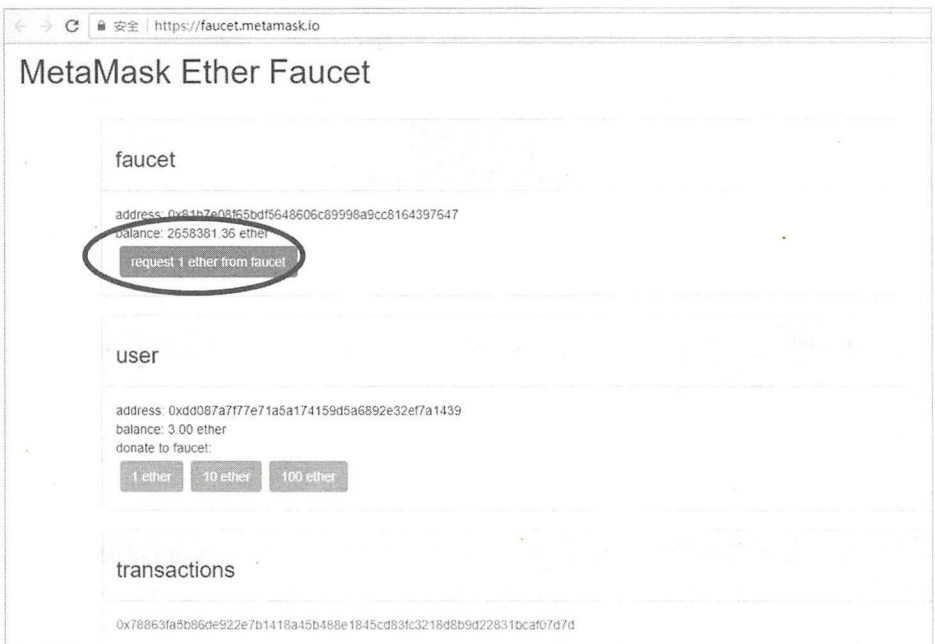


图 7.31　MetaMask 领取以太币

（7）获得测试用的以太币后，在账户信息上即可显示出来。例如，在如图 7.32 所示的测试账户中，初期获得了测试用的 3 个以太币，在测试智能合约部署时消耗了 0.001 个以太币，剩余 2.999 个以太币。

仅供非商业用途或交流学习使用

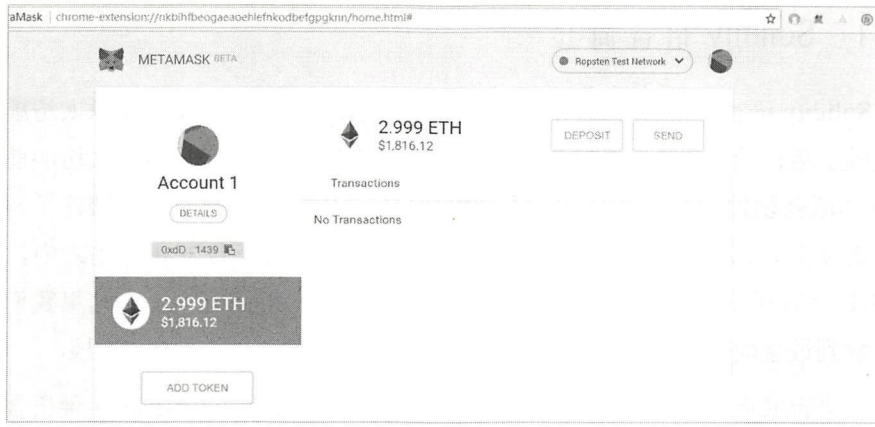


图 7.32　MetaMask 测试账户的以太币余额

TIPS:

在账户信息中单击 DETAILS（详情）按钮，会弹出账户的二维码、详情（通过 Etherscan 网站查看），并导出账户私钥。在输入账户创建密码后，系统会显示私钥，用户需将之稳妥、秘密保存，避免丢失或被黑客窃取；否则就会失去该账户中拥有的真实的数字货币资产。

当然，在测试网络下，当我们拥有了测试用的以太币后，在开发智能合约时，可以很方便地在测试网络上进行部署和测试，不再受到部署过程需要消耗 Gas 的影响。但需要提醒大家的是，如果我们仅使用测试网络取得了以太币，那么，当转换到以太坊主网络后，可以看到账户中真实的以太币数量仍然为 0。要在以太坊主网络中拥有以太币，需要挖矿或购买。

7.7　以太坊开发的编程语言 Solidity

上一节介绍了以太坊开发的常用平台或者说客户端，它们相当于一座宏伟建筑的不同入口，而这座宏伟建筑是怎样建造起来的呢？所需的基本“材料”——编程语言目前也有多种，如 C++、Golang、Solidity 等，其中最常用的就是 Solidity 语言。本节我们正式“登堂入室”，对 Solidity 语言进行深入的讲解。

仅供非商业用途或交流学习使用

7.7.1 Solidity 语言简介

Solidity 语言是一种专门用于编写和执行智能合约的语言，是在以太坊虚拟机基础上运行的、面向合约的高级语言，最初是在 2014 年 8 月由以太坊的前任 CTO 和联合创始人 Gavin Wood 提出来的，后来由以太坊开发人员组建了一支专门的团队，对 Solidity 语言进行不断改进，目前仍在开发和优化之中，在 GitHub 上的开发存储区域是 https://github.com/ethereum/solidity，在这里我们可以了解到最全面的关于 Solidity 语言开发和迭代的过程详情、相关文档。

在语言的风格上，Solidity 语言受到 C++、Python 和 JavaScript 3 种语言的深刻影响，它是一种静态类型的编程语言，以字节码（Bytecode）的模式进行编译，因此可以在以太坊虚拟机上运行。Gavin Wood 在开发 Solidity 语言时借鉴了 JavaScript 的 ECMAScript 脚本语言的语法规则，使它与现有的网页开发语言有些类似，但其实有较大不同，如 Solidity 语言拥有静态类型、可变返回函数等。最重要的一点是，Solidity 语言可以编写具有自执行的业务逻辑、嵌入智能合约中的合约，因此它不但是以太坊的基础编程语言之一，而且是其他绝大部分基于以太坊的、具有智能合约的各种区块链产品（Blockchain 2.0）的基础编程语言，被广泛应用于目前绝大多数区块链产品，如超级账本（Hyperledger）项目就是用 Solidity 语言开发而成的。

目前最好的 Solidity 语言开发客户端是 Remix 这种网页模式的集成开发环境，在上一节中已经介绍了，在此不再详述。

7.7.2 Solidity 语言的常用语句

首先通过一个用 Solidity 语言编写的智能合约的例子来初步了解 Solidity 语言的一些常用语句。这是一个用于投票的智能合约[①]，程序代码如下：

```
pragma solidity ^0.4.22;

/// 标题：代表投票
contract Ballot {
```

①本章内容参考 https://solidity.readthedocs.io/en/latest/。

仅供非商业用途或交流学习使用

```
    // 在此处声明一个新的复合类型，用于下一步的变量，代表一个单一的投票者
    struct Voter {
        uint weight;          // 投票累计
        bool voted;           // 如果为真，则该人已投票
        address delegate;     // 投票人投向的地址
        uint vote;            // 投票提案索引
    }

    // 用于单一提案的结构类型
    struct Proposal {
        bytes32 name;         // 短名称(最多 32 字节)
        uint voteCount;       // 累计投票数
    }

    address public chairperson;

    // 在此处声明一个状态变量，为每个可能的地址存储“Voter”结构
    mapping(address => Voter) public voters;

    // 定义一个动态大小的数组，元素类型为“Proposal”结构
    Proposal[] public proposals;

    /// 选择“proposalNames”创建新投票
    constructor(bytes32[] proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // 对于每个提供的提案名，创建一个新的提案对象，加到数组末尾
        for (uint i = 0; i < proposalNames.length; i++) {
            // “Proposal({...})”创建一个临时提案对象，并补充到“proposals”末尾
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // 对本次投票中的“Voter”给予投票权，只能由“chairperson（主席）”调用
    function giveRightToVote(address voter) public {
        // 如果“require”的第一个参数结果为“false”，则执行中止，所有状态变化、
以太币余额变化都会还原
        // 在老版 EVM 中，这会耗光所有 Gas 费用，但现在已经不这样了
        // 使用“require”来检查函数是否正确调用是一个好方法
```

仅供非商业用途或交流学习使用

```
        // 使用第二个参数给出了出错的原因

        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// 将投票权授予投票人“to”
    function delegate(address to) public {
        // 对引用进行赋值
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // 当“to”也是被授权时，继续转授权
        // 一般来说，这样的循环很危险，因为它们的运行时间过长，与完成一个区块相比
需要更多的 Gas 费用

        // 在本例中，代表授权不会执行。但在其他情况下，这一循环可能会导致合约非正
常完成
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // 检验是否存在不允许的代表授权
            require(to != msg.sender, "Found loop in delegation.");
        }

        // 由于“sender”是一个引用，所以修改了“voters[msg.sender].voted”
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // 如果代表已投票，则直接增加投票数量
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
```

仅供非商业用途或交流学习使用

```
            // 如果代表未投票，则增加其权重
            delegate_.weight += sender.weight;
        }
    }

    /// 将你的投票（包括授权给你的投票）投给提案 "proposals[proposal].name"
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // 如果 "proposal" 超过数组范围，则将自动释放这一信息，并还原所有改变
        proposals[proposal].voteCount += sender.weight;
    }

    /// 根据此前全部投票，统计获胜提案
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // 调用 Calls winningProposal()函数取得包含在提案数组中的获胜者索引，并返
回获胜者名称
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

从上述程序代码中可以总结出 Solidity 语言的一些常用语句。

1. 版本号指示

每个 Solidity 程序都以版本号指示（Version Pragma）开头，一般情况下的格式如下：

仅供非商业用途或交流学习使用

```
pragma solidity ^0.4.22;
```

在源程序的开头先标明该源程序所使用的 Solidity 版本号。采用这种方式的原因是 Solidity 语言仍然在不断地迭代，每次新版本的发布都会增加新的功能，但同时也可能带来对原来在老版本下编制的源程序编译的不兼容问题，所以通过在源程序的开头标明版本号，以减少这种语法规则改变可能导致的不兼容现象。

比如，在本例中标明了它是用 Solidity 0.4.22 版本编写的，那么我们就不宜采用低于这一版本的编译器来编译，否则可能会出现不兼容现象。那么，用高版本的编译器来编译，是否就必然没有不兼容问题呢？这也不一定，因为根据版本号的制定规则，某种语言从 0.4.22 发展到 0.5.0、0.6.0 版本，可能不会有大的、革命性的变化，以此来编译 0.4.22 的源程序没有问题；但是如果发展到 4.0.0 版本，则一定发生了巨大的变化，以此来编译 0.4.22 的源程序依然可能会产生不兼容问题。

2. 导入其他源文件

与 JavaScript 相似，Solidity 语言支持“导入声明”，其通用形式一般是这样的：

```
import "filename";
```

其功能是将文件名为“filename”的源文件中的全局符号导入现有的程序文件的全局符号域中。

与此相似的是对导入声明的定制化应用，如下：

```
import * as symbolName from "filename";
```

这条语句的功能是：创建一个新的全局符号（symbolName），其组成元素全部来自文件名“filename”中的全局符号。与这条语句等价的表述是：

```
import "filename" as symbolName;
```

更进一步、更细化的导入声明采用如下语句：

```
import {symbol1 as alias, symbol2} from "filename";
```

这条语句的含义是：从“filename”源文件中导入两个全局符号 symbol1 和 symbol2 到现有文件中，其中 symbol1 在现有文件中重命名为 alias。

仅供非商业用途或交流学习使用

另外，在“filename”中，可以使用由右斜杠“/”连接的文件路径表述方式，比如 github/xxx/yyy.sol 这样的形式。如果是相对路径，则用“.”表示当前文件夹，用“..”表示父文件夹（上一级文件夹）。如果导入的是当前文件夹下的某个文件，则需要用 import ./x as x，而不能直接用 import x as x，因为在后面的这条语句中，导入的 x 文件会被定位到另一个全局性的文件夹中，而不是当前文件夹。

TIPS:

> 导入的源文件，除在本机上的文件之外，也可以映射到其他存储区域，比如 GitHub 上的源文件。

在 Remix 模式下，可以自动重定向到 GitHub 区域，也可以通过网络自动同步来找回丢失或损坏的文件，具体语句如下：

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as
it_mapping;
```

3. 注释

注释部分是编程达到可读性的重要组成部分，通过在源程序中加上注释语句，对代码的某个部分的功能或其他方面进行简单的解释，有利于自己在日后对程序进行修订和完善，也有利于其他程序员对程序中的成千上万行代码进行阅读和理解，从而更好地进行完善、迭代。

简单来说，注释分为两种：一种是单行的解释，以右双斜杠“//”开头，一句话就说完；如果注释很长，一句话说不完而要分成几句话来说，那么单行注释符号就不够用了，这时需要采用多行注释模式，注释符号为“/*...*/”，通常斜杠和星号各占一行，在星号之间的多行注释可分行录入，最后用一个“*/”结尾即可，编译时系统会将这些符号之间的代码视为注释内容。

例如：

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

TIPS:

在使用 Solidity 语言编程时，有的程序员会用到“三斜杠”的注释，或者“/**.....*/”形式，它们可以直接使用，其功能与传统注释符号完全一样，如“三斜杠”就是单行注释的符号，同理“/**”意味着多行注释。

除以上 3 部分之外，一个 Solidity 源程序还包括用 Solidity 写好的各行代码。由于不同的智能合约产生的功能不同，所以具体的代码也不相同，每条语句都有各自的特点和功能，研究时只能针对具体的案例进行分析，这里就没有再对源代码进行“划片分类”，我们将在后面的内容中结合某一具体的源程序逐行解读源代码。

7.7.3 Solidity 程序初步解读

在本小节中，我们将进入一个 Solidity 程序的内部，对常见的重点语句进行解读。

这是一个最简单的、模拟数字加密货币发行过程的 Solidity 示例程序。首先，凭空创建一些数字货币，当然这一功能只有数字合约的创建者才能使用；其次，任何人只要掌握了私钥和公钥这一密钥对，在无须注册账户和密码的情况下即可互相使用该数字货币进行转账。具体代码如下：

```
pragma solidity ^0.4.21;

contract Coin {
    // 关键字“public”令这些变量从外部可读
    address public minter;
    mapping (address => uint) public balances;

    // “事件（event）”允许轻客户端对变化进行快速反应
    event Sent(address from, address to, uint amount);

    // 以下为构造器，其代码只在合约创建时运行
    function Coin() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
```

```
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

在该程序中，我们首先看到的是版本号指示语句“pragma solidity ^0.4.21;”；其次可以发现，在注释中主要使用的是单行注释语句，由“//”引导。

合约的命名采用“contract Coin {....}”的语句引导，将本合约命名为 Coin，在花括号内的语句都是用来实现 Coin 的各项功能的。

我们进入 Coin 合约的内部，可以看到第一行语句是“address public minter;”，这条语句声明了一个地址类型的状态变量，该变量可以公开（public）访问。地址（address）类型是一个 160 位的数据类型，不可以用于算术计算，只能用于存储合约地址、外部用户的密钥对（私钥、公钥）等数据。关键字 public（公开）则自动为这一地址增加了一项功能：可以从合约外部对此地址的当前值进行访问。

第二行语句是“mapping (address => uint) public balances;”，它也创建了一个公共的状态变量 balances，但这是一个更加复杂的数据类型，将地址映射为一个无符号整数（uint）。这一映射其实功能比较有限，而且增加了复杂性，所以在不必要的时候尽量少用。另外，在 Solidity 语言中，一旦应用了 public 属性，则可以应用“取值函数（Getter Function）”，功能更强大，其形式如下所示：

```
function balances(address _account) public view returns (uint) {
    return balances[_account];
}
```

使用这一函数，可以很容易地提取到某个账户的资产状况数值。

第三行语句是“event Sent(address from, address to, uint amount);”，在这条句中声明了一个“事件（event）”，它是由上述程序中最后一个函数 function send()

所释放的。当合约被部署后，系统会收听所有的、由函数在区块链上释放的“事件”，而每个释放的事件都带有参数——从哪儿来（from）、到哪儿去（to）、支付多少钱（amount），这样该交易很容易被区块链上的监听节点追踪到，从而实现了区块链的全网确认功能。

接下来是一个 Coin()函数。

```
function Coin() public {
        minter = msg.sender;
    }
```

Coin()这个特殊的函数是一个构造器，只在合约创建时运行，它用 msg 函数永久存储合约创建者的地址，以及用来访问区块链的一些属性。其中，msg.sender 中存储的是当前函数被调用的地址。

上述示例的合约最后以两个函数结束，这些函数可以由用户或合约调用，本例中包括 mint()和 send()函数。mint()函数只能由合约创建的账户调用，由其他账户调用无效；send()函数则可以由取得代币的任何账户调用，用来向其他任意账户转账。

chapter EIGHT

第8章 Solidity 开发基础

随着区块链技术向全社会各方面的全方位渗透，以太坊网络以其可编程、开放性强的特点成为区块链技术开发的主流平台。此外，在 Solidity 语言的不断迭代和日益成熟的基础上，大量的新进入者以 Solidity 语言为工具，开发出了各类 DApp（Decentralized Applications，去中心化应用程序），目前主要包括金融、交易、游戏、健康、公益、媒体、社交、保险、权益、博彩、供应链、投票等数十个行业或业务范畴。

我们在前几章中系统地介绍了区块链、比特币、以太坊编程的各方面知识，最终是为了在对区块链和以太坊充分了解的基础上进入应用开发的实际操作阶段。从本章开始，我们将专注于了解和掌握以太坊开发的流行工具——Solidity 语言。

以太坊应用的编程开发在本质上还是一种编程语言的应用，从计算机语言的角度来看，它依然会涉及数据结构、算法等基本原理。当然，由于是建立在区块链的原理和基础上的，所以以太坊应用的基础原理与传统编程相比有其独特之处。本章将以 Solidity 语言为通道[①]，深入了解以太坊应用开发的基础元素，因为只有打下了坚实的基础，才能走得更远。

① 本章主要内容参考 Solidity 语言官方文档 http://solidity.readthedocs.io/en/latest/。

8.1 Solidity 语言的数据类型

Solidity 语言是一种静态类型的编程语言，这种语言的特点是在编程中需要指定每个变量的类型，因此，在使用 Solidity 语言开发以太坊应用时，需要首先了解 Solidity 语言的数据类型。Solidity 语言提供了几种基本的数据类型，还可以通过基本类型进行组合形成复合类型。

8.1.1 数值类型

数值类型是所有编程语言的基础类型，Solidity 语言当然也不例外，它主要有两种数值类型。

1. 整数类型

整数分为两种类型：各种长度的有符号整数（int）和无符号整数（uint），比如从 8 位到 256 位的有符号整数，表示为 int8 和 int256，无符号整数与此类似。

在 Solidity 语言中，默认的整数位数为 256 位，所以直接分别用 int 和 uint 默认表示 int256 和 uint256 型的整数。

整数之间的计算包括三大类型，具体如表 8.1 所示。

表 8.1 整数的计算类型

计算类型	计算符号	计算功能
大小比较（6 个）	<=	小于等于
	<	小于
	==	等于
	!=	不等于
	>	大于
	>=	大于等于
位运算（4 个）	&	按位与
	\|	按位或
	^	按位异或
	~	按位求非

续表

计算类型	计算符号	计算功能
算术运算（10 个）	+	加
	++	单目加
	-	减
	--	单目减
	*	乘
	/	除
	%	求余
	**	乘方
	<<	左移位
	>>	右移位

左移位 x<<y 相当于 x*2**y，右移位 x>>y 相当于 x/2**y。如果移位次数为负，则将产生运行时间异常。另外，与其他编程语言略有不同的是，在 Solidity 语言中，右移位映射为除法，所以对负数右移位通常向零舍入为 0；而在其他编程语言中，对负数右移位与“向下舍入的除法”相似，其舍入结果是负无穷。

2. 逻辑类型（布尔值）

布尔值（bool）只有两个常量值：true（真）和 false（假）。

布尔运算符有 5 种，如表 8.2 所示。

表 8.2　布尔运算符

符　　号	计算功能
!	逻辑非
&&	逻辑与
\|\|	逻辑或
==	等于
!=	不等于

运算符“&&”和“||”常用作“短路运算符”，比如 f(x)&&g(y)，当 f(x)为 false 时，无须再计算或关注 g(y)是否为真或为假，这条语句的结果一定为 false。同理，对于 f(x)||g(y)，如果 f(x)为 true，则无须再计算 g(y)是为 true 或 false，这条语句的结果一定为 true。这样就形成了对整条语句的快速计算，称为“短路”计算。

8.1.2 地址类型

地址类型以 address 表示，根据以太坊的地址规则，它的长度有 20 位。地址是所有智能合约的基础，同时地址类型也包含成员变量。

运算符包括<=、<、==、!=、>=、>共 6 类。

地址类型的成员变量有两个：balance（账户）和 transfer（转账）。

地址类型的相关函数如表 8.3 所示。

表 8.3 地址类型的相关函数

函数表达式	函数功能
<address>.balance (uint256)	返回以“伟（wei）”为单位的该地址的以太币账户余额
<address>.transfer(uint256 amount)	给某个地址转入指定数量（以 wei 计数）的以太币，2300 个 Gas 的费用，不可调整
<address>.send(uint256 amount) returns (bool)	给某个地址转入指定数量（以 wei 计数）的以太币，2300 个 Gas 的费用，不可调整；如果失败则返回逻辑值 false
<address>.call(...) returns (bool)	发布一个低级别的调用（call），如果失败则返回逻辑值 false，使用可用的全部 Gas 费用（可以调整）
<address>.callcode(...) returns (bool)	发布一个低级别的代码调用（callcode），如果失败则返回逻辑值 false，使用可用的全部 Gas 费用（可以调整）
<address>.delegatecall(...) returns (bool)	发布一个低级别的代表调用（delegatecall），如果失败则返回逻辑值 false，使用可用的全部 Gas 费用（可以调整）

地址的属性 balance 用来查询或索引一个地址的账户余额，然后可以用 transfer()函数给某个地址发送以太币（以 wei 为单位），例如以下这段代码：

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

其含义是：如果 x 地址中的以太币余额低于 10，“我的账户”中的以太币余额高于 10，就向 x 账户中转入 10 个以太币。在这段代码中，x 作为合约账户，其转账（transfer）命令是由以太坊虚拟机自动执行的，而且不可阻止。如果由

于余额不足、Gas 费用不足而导致合约不能执行，则已经执行的部分会回滚（恢复原状），当前合约会中止并发送“异常”消息。

地址的另一个成员变量是 send（发送）。send 是 transfer（转账）的低级别对应命令，如果执行失败，则合约不会发送“异常”消息并中止，但 send 会返回逻辑值 false。

TIPS:

在使用 send 时存在风险：如果调用堆栈超过 1024KB，或者接收方的 Gas 耗尽，转账就失败了。所以，为了实现安全转账，要么在使用 send 时先检查返回值，要么采用函数 transfer()，要么设定一种更好的模式：一旦转账失败，接收方要退回已经接收到的以太币。

上文已经提到，在 Solidity 语言中，address 有 3 个调用函数：call()、callcode()和 delegatecall()，其中 callcode()函数将来会从 Solidity 语言中删除，所以应尽量少用。call()这个调用函数是 address 中最常用的调用函数，可以从命令行参数中取得任意要取的值，其返回值是布尔值：当返回 true 时，表示调用正常，调用函数中止；当返回 false 时，表示发生了以太坊虚拟机异常。但是 call()函数并不返回实际调用的数据值。

call()函数也有两个次级函数：gas()和 value()，分别用来修改合约在编译过程中所支付的费用（Gas）和以太币值。这两个函数可以单独使用，也可以合在一条语句中使用，对先后顺序并无要求。例如：

```
namReg.call.gas(1000000)("register", "MyName");
nameReg.call.value(1 ether)("register", "MyName");
```

这两条语句的功能等同于以下语句：

```
nameReg.call.gas(1000000).value(1 ether)("register", "MyName");
```

另一个函数 delegatecall()的使用方式与 call()函数的使用方式相似，唯一的区别是给定的地址的代码不同，其他信息都是从当前合约中提取的，但是 delegatecall()函数使用的是存储在另一个合约中的库代码（Library Code），所以用户在使用 delegatecall()函数之前要确认两个合约的存储设计都适用于该函数。

这 3 个函数都是低阶函数，在使用时要非常小心，因为有可能调用的未知

合约是恶意合约，当你调用它时相当于交出了本合约的控制权，这时被调用的恶意合约可能反过来调用你的合约。因此，当调用函数返回时，要防备它改变了原合约中的参数的性质。

8.1.3 字节数组

字节数组包括两种类型：固定长度的字节数组和可变长度的字节数组。

1. 固定长度的字节数组

固定长度的字节数组用bytes1,bytes2,bytes3,…,bytes32来声明，共有32个，其中用byte代表bytes1。

bytes1只能存储1字节，也就是二进制8位的内容；bytes2只能存储2字节，也就是二进制16位的内容；bytes3只能存储3字节，也就是二进制24位的内容；以此类推，bytes32只能存储32字节，也就是二进制 $32 \times 8 = 256$ 位的内容。

例如下面的程序：

```
pragma solidity ^0.4.4;

contract C {

    // 全部地址长度:0x6c697975656368756e

    byte public a = 0x6c;
    bytes1 public b = 0x6c;
    bytes2 public c = 0x6c69;
    bytes3 public d = 0x6c6979;
    bytes4 public e = 0x6c697975;

    // ...
    bytes8 public f = 0x6c69797565636875;
    bytes9 public g = 0x6c697975656368756e;

}
```

固定长度字节数组的运算符包括3类，如表8.4所示。

表 8.4　固定长度字节数组的运算符

类　型	运 算 符
比较运算符	<=、<、==、!=、>、>=，共 6 个
位运算符	&（按位与）、\|（按位或）、^（按位异或）、~（按位求非）、<<（左移位）、>>（右移位），共 6 个
索引访问	如果 x 是一个 bytesI，则 $x[k]$（当 $0<k<I$ 时）返回第 k 位的数值（只读）

固定长度字节数组的成员函数只有一个：.length，通过这个函数可以取得字节数组的固定长度（只读），返回值的单位是字节。

2. 可变长度的字节数组

可变长度的字节数组有两类，包括：①bytes，即由可变长度的字节组成的数组，与数组（array）的性质相同；②字符串（string），由可变长度的 UTF-8 字符组成的字符串，相当于一种特殊数组，不能调用.length 或索引来访问其元素。

后文有对数组的专门介绍，在此不再详述。

8.1.4　地址常量

地址常量是一个十六进制的数字，以 0x 开头，如 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF。当作为地址常量时，需要通过地址常量的“求和校验”，不能通过“求和校验”的，或者长度为 39～41 位的十六进制常量，都会产生一个警告，并被当作常见的有理数进行处理。

8.1.5　有理数和整数常量

整数常量由 0～9 个数字按一定的顺序组成。由于在 Solidity 语言中未引入八进制，因此没有以 0 开头的整数常量。

小数常量由小数点和数位组成，在小数点左右两侧要至少一侧有数字，如“1.”“.2”“1.3”，这些都是合乎规则的小数常量。

科学计数法也得到支持，但是基数可以是小数，而指数部分不能为小数。比如，以下为正常可用的科学计数法：

```
2e10, -2e10, 2e-10, 2.5e1
```

仅供非商业用途或交流学习使用

只要不转换成非常量类型，数据常量可以实现任意精度，因此数据常量的计算不会溢出，当数据常量表达式用作除法时也不会被截断。例如，在(2**800+1)2**800 这个表达式中，本来 2**800 由于超过了 Solidity 的数据长度范围应该触发溢出报警，但由于结果是 1（这是一个 uint8 类型的数据常量），所以该表达式仍然能够编译通过。另外，“.5*6”的结果是数据常量 3，而不是 3.0，尽管在计算过程中使用了“.5”这样一个非数据常量。



对于数据常量来说，只要运算的数据是整数型常量，那么整数型变量的全部运算符都可以应用于数据常量；但是当运算的数据中有一个是小数时，位操作计算就不允许使用；当指数为小数时，乘方计算也不允许使用。

在 Solidity 语言中，每个有理数都有一个数据常量类型，整数常量与有理数常量都属于“数据常量”，此外所有的表达式也都属于数据常量，比如 1+2、2+3 这样的表达式属于同一类数据常量，因为它们都有 3 个有理数。

需要注意的是，在数据常量表达式中如果使用了非数据常量表达式，就转换为非数据常量表达式，不管其值是否还是数据常量。例如，下面的代码就不能编译通过：

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

尽管我们知道 a=1，表达式 b 的结果也等于一个整数，但由于 b 的表达式中存在一部分非数据常量表达式“2.5+a”，因而被转换为非数据常量表达式，而这又与该句的属性定位（uint128，一种数据常量）产生了类型冲突，无法通过 Solidity 的类型检测，从而导致编译无法通过。

8.1.6 字符串常量

字符串常量可以用单、双引号来表示，如"football"、'bar'均可。与 C 语言不同的是，Solidity 语言不会自动在位值后补充 0 来结尾，比如“bar”的位数就是 3 位。和整数常量一样，字符串常量的类型也是可以转换的，根据具体情况和条件，字符串常量通常会转换为 bytes1,bytes2,…,bytes32 这类字节数组，或者字符串数组。

仅供非商业用途或交流学习使用

字符串常量支持转义字符，如\xNN 和\uNNNN。其中，\xNN 表示选取一个十六进制的值，并插入适当的字节；\uNNNN 表示选取一个 Unicode 代码点，并插入一个编码方式为 UTF-8 的序列。

8.1.7　十六进制常量

十六进制常量以前缀字母 hex 为标志，用单引号或双引号括住，如 hex "001122FF"，引号中的内容必须为十六进制字符，其值是采用二进制表示的值。

十六进制常量与字符串常量的性质相同，转换限制也相同。

8.1.8　枚举类型

枚举类型是 Solidity 语言中创建用户自定义类型的一种手段，其格式为：enum+枚举类型的名称+{枚举成员 1,枚举成员 2,枚举成员 3,…}。

枚举类型可以采用显式转换的方式与整数类型之间进行相互转换，但不允许隐式转换。在显式转换时，Solidity 将进行值域校验，如果出现超域现象就触发失败。在枚举类型中至少要有一个成员。下面是一段枚举类型的示例代码：

```
pragma solidity ^0.4.16;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

仅供非商业用途或交流学习使用

8.1.9 函数类型

函数分为两类：内部函数（internal）和外部函数（external）。

内部函数只能在当前合约内部调用，更具体的一些内部函数甚至只能在当前代码段内部使用，比如内部库函数、继承函数等。原因是内部函数不能在当前合约的运行环境之外运行，要调用内部函数，需要跳转到它的入口标签，就像在当前合约内部调用一个函数一样。

外部函数由地址、函数名组成，可以使用外部函数调用来传递数据、返回结果。

函数类型采用如下表示方式：

```
function (<parameter types>) {internal|external} [pure|constant|view|
payable] [returns (<return types>)]
```

其中，参数类型（<parameter types>）可以为空，但是[returens (<return types>)]这部分不能为空。当然，如果某个函数不返回任何值，那么[returens (<return types>)]这部分内容需省略掉。

TIPS:

> 在 Solidity 语言中，函数的默认类型是内部函数，所以关键字 internal 可以省略。与之相对的是合约函数，它们默认是公共的，只有当按其类型名称使用时，才默认是内部函数。

在当前合约中，有两种方式来访问函数：①直接使用函数名称，如“function”，表示访问的是内部函数；②使用“this.function”的方式，表示访问的是外部函数。

如果某个函数类型变量没有初始化，或者在删除（delete）之后又调用它，则都会导致系统发出异常信息。

外部函数在 Solidity 的外部环境中使用时与内部函数并无二致，因为 Solidity 将其地址、函数标识符整合编码进一个用 bytes24 声明的字节数组中，使其编译不受影响。

在当前合约中，如果一个函数被声明为公共类型（public），那么它既可以

仅供非商业用途或交流学习使用

作为内部函数使用，也可以作为外部函数使用，区别仅在于使用的方式：如果直接用函数名来调用，那就是内部函数；如果用类似“this.function”的方式来调用，就是把它当作外部函数在使用的。

另外，对于外部函数、公共类型的函数来说，它们都有一个特殊的成员函数——selector（选择器），它返回一个“应用程序二进制接口”的选择符，即该调用的前 4 字节的数据，用来对调用的函数进行“精准定位”，其代码如下：

```
pragma solidity ^0.4.16;

contract Selector {
  function f() public view returns (bytes4) {
    return this.f.selector;
  }
}
```

下面通过一段示例代码来展示对内部函数的使用，如下：

```
pragma solidity ^0.4.16;

library ArrayUtils {
  // 内部函数可用于内部库函数，因为它们是同一代码环境的组成部分
  function map(uint[] memory self, function (uint) pure returns (uint) f)
    internal
    pure
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint, uint) pure returns (uint) f
  )
    internal
    pure
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
```

仅供非商业用途或交流学习使用




```
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal pure returns (uint[] memory r)
{
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
  function pyramid(uint l) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal pure returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}
```

对于外部函数的使用，下面的程序提供了示例：

```
pragma solidity ^0.4.22;

contract Oracle {
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
  function query(bytes data, function(bytes memory) external callback)
public {
    requests.push(Request(data, callback));
    emit NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) public {
    // 在此处进行检验，产生来自可信源的回复
```

仅供非商业用途或交流学习使用




```
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // 已知的合约
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) public {
    require(
        msg.sender == address(oracle),
        "Only oracle can call this."
    );
    // 使用数据
  }
}
```

8.1.10　数据位置

在 Solidity 语言中有一些复合的数据类型，如数组（array）、结构（struct）等，它们都有一个额外的标注，称为“数据位置”，用来标注这些数据存储在哪里，是存储在内存（memory）中，还是存储在存储介质（storage）如硬盘中？根据编译环境不同，可能有默认的存储位置，但是可以通过在数据类型之后添加 memory 或 storage 来重新指定存储位置。函数的参数（包括返回的参数）默认是存储在内存中的，而局部变量、状态变量则默认存储在存储介质中。

数据位置这件事相当重要。当存储介质与内存之间进行变量赋值，或者从存储介质赋值到状态变量时，会创建独立的复制过程，即使源数据被删除了，也不会影响独立的复制过程所产生的赋值结果；而本地存储变量之间的赋值其实只赋了一个引用，这个引用总是指向该状态变量，即使该状态变量发生了变化，也不会改变引用状态。内存变量之间的赋值并不创建独立的复制过程，如果原值被删除了，那么被赋值的变量就会出错。下面是一段示例程序：

```
pragma solidity ^0.4.20;

contract C {
    uint[] x; // x 的数据位置在存储介质中
```

仅供非商业用途或交流学习使用




```
    // 作为函数参数的 memoryArray，数据位置在内存中
    function f(uint[] memoryArray) public {
        x = memoryArray; // 正常，将整个数组复制到存储介质中
        var y = x;       // 正常，赋值一个指针，y 的数据位置在存储介质中
        y[7];            // 返回数组的第 8 个元素
        y.length = 2;    // 通过 y 的赋值，修改了 x
        delete x;        // 清除 x 数组，y 不受影响（独立复制）
        // 以下语句不能运行，因为虽然调用了 g()和 h()两个函数，但是这两个函数对于
x 的引用不变，仍指向 x 数组，而 x 数组已经被清除了
        g(x); // 调用 g()函数，产生一个对 x 的引用
        h(x);
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {}
}
```

8.1.11 数组

数组的大小在编译时是定值，但在其他时间可以是一个动态值。数据位置在存储介质上的数组，其元素类型可以是任意的；但对于数据位置在内存中的数组来说，它的元素不能是映射（mapping）类型的，而且如果是公共可见函数的参数，那么这些元素必须是二进制类型的。

数组的表示形式是 T[k]，其含义是数组类型为 T，固定长度为 k；如果数组长度是动态的，则表示为 T[]。例如，一个有 5 个动态长度的 uint 数据的数组，表示为 uint[][5]；要访问第三个动态数组的第二个元素，则使用 x[2][1]，因为数组的下标都是从 0 开始的。

bytes（字节）数组和 string（字符串）数组都是特殊类型的数组。当 bytes 数组单独出现时，它的含义与 byte[]相似，但在存储时 bytes 数组采用的是紧凑模式，所以从费用节约的角度出发，应该多使用 bytes 数组而少使用 byte[]。string 数组与 bytes 数组相似，但是 string 数组不能用长度或索引值来访问其中的元素。根据经验，bytes 数组中存放的是任意长度的原始字节数据，而 string 数组中存放的是任意长度的字符（UTF-8）数据。但是如果限定固定数组的长度，则最好根据具体的长度，对应使用 bytes1～bytes32。

仅供非商业用途或交流学习使用



在设置数组时，可以将其属性设置为 public（公共），这样 Solidity 就可以创建一个取数器，这个取数器的必不可少的一个参数就是数值的索引。

数组有两种特殊类型。

（1）内存配置数组。在内存中创建可变长度的数组可以使用“new”关键字。与存储在存储介质中的数组不同，内存数组一旦创建，不能通过给成员函数“.length”赋值来改变内存数组的知识结构。

（2）数组常量或内联数组。数组常量是写成表达式的数组，不能马上赋值给其他变量。它是固定长度的内存数组，所给元素的类型就是数组常量的基本类型。例如[1,2,3]这样一个数组常量，因为它的 3 个元素都是 uint8 型的，所以整个数组就是 uint8[3] memory 类型的。在编程中，这类数组的第一个元素按要求要转换为 uint 型，即[uint(1),2,3]。另外，在内存中，固定长度的数组常量不能赋值给动态长度的数组，比如下面的代码就不能编译通过：

```
pragma solidity ^0.4.20;

contract C {
    function f() public {
        // 下一行发生类型错误，uint[3] memory 数组不能转换为动态内存数组 uint[]
memory
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

数组的成员函数有 3 个。

（1）length（长度）。length 成员函数持有数组的元素个数，通过改变动态存储数组的成员函数.length，可以重新确定其元素个数；而内存数组一旦创建，其长度就是固定的。

（2）push（写入）。动态存储数组、bytes 数组（注：string 数组不适用）都有一个成员函数——push，可以在原数组的末尾增加新的元素，并返回数组的新长度。

（3）pop（移除）。动态存储数组、bytes 数组（注：string 数组不适用）都有一个成员函数——pop，其功能是在数组尾部删除一个元素。

仅供非商业用途或交流学习使用



关于数组的使用，有以下注意事项。

（1）在外部函数中，目前还不能使用“元素为数组的数组”。

（2）由于以太坊虚拟机的限制，目前外部函数调用不能返回动态数组。例如，在语句“contract C { function f() returns (uint[]) { ... } }”中的函数 f()，它返回的并不是 Solidity 的动态数组内容，而是从 web3.js 平台上调用的内容。对于这一点的变通做法是使用大规模的静态长度的数组，如下面的程序代码所示：

```
pragma solidity ^0.4.16;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // 注意，以下并不是一对动态数组，而是一个双元素的动态数组（如两个长度的固定大小数组）
    bool[2][] m_pairsOfFlags;

    // newPairs 存储在内存中，默认用于函数参数存储
    function setAllFlagPairs(bool[2][] newPairs) public {
        // 对存储数组赋值，取代完全数组
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // 访问不存在的索引会产生异常
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // 如果新的空间太小，则无关的数组元素将被清除
        m_pairsOfFlags.length = newSize;
    }

    function clear() public {
        // 这些命令将完全清除数组
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // 等效表述
        m_pairsOfFlags.length = 0;
    }
```

仅供非商业用途或交流学习使用




```
    bytes m_byteData;

    function byteArrays(bytes data) public {
        // 字节数组不同，其存储时不补空，但视同 uint8[]对待
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = byte(8);
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) public returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) public pure returns (bytes)
{
        // 使用“new”关键字创建动态内存数组
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // 创建动态字节数组
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(uint8(i));
        return b;
    }
}
```

8.1.12　结构

前面所讲的都是 Solidity 中的基本类型，但 struct（结构）是 Solidity 中的自定义类型，通过使用 Solidity 的关键字 struct 进行自定义，在结构体内还可以再包含字符串、整型、映射、结构体等复杂类型。

结构可以在映射、数组中使用，而且可以再包含映射、数组元素。下面是一个合约的代码，其中包含结构的应用。

```
pragma solidity ^0.4.11;

contract CrowdFunding {
    // 定义一个新变量，它有两个定义域
    struct Funder {
        address addr;
```

```
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) public returns
(uint campaignID) {
        campaignID = numCampaigns++; // campaignID是返回变量
        // 创建新结构并存储，忽略映射类型
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // 创建一个新的临时内存结构，并用给定变量值进行初始化，复制到存储器中。注
意:也可使用 Funder(msg.sender, msg.value)来初始化
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount:
msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool
reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

需要注意的是，尽管结构自身可以是一个映射成员的值，但在结构的成员

中不能再包含与自己类型相同的成员，因为这种嵌套会导致结构的大小失控，而系统对于结构的要求是：结构的大小必须是明确的和有限的。

在所有的函数中，结构类型赋值到局部变量的方式并不会复制结构的内容，而只会存储一个引用，将每个局部变量成员的赋值写入状态变量中。当然，结构的成员变量也可以被直接访问，无须先对其赋值一个局部变量，比如“campaigns[campaignID].amount = 0”这样一条语句是成立的。

8.1.13　映射

映射（mapping）类型通过“mapping(_KeyType => _ValueType)”这种方式来声明，其中_KeyType 可以是任意变量类型，如动态长度的数组、合约、枚举、结构等，但不能是映射自身这种类型；而_ValueType 可以是任意变量类型，包括映射在内。

映射可以被视作一个哈希表，所有可能的键会以虚拟化形式创建，映射到这个类型的默认值中（此时，该类型的默认值用二进制表示，二进制的各位全部为 0）。在映射表中，并不存储键的数据，仅仅存储它的 Keccak-256 哈希值，这个哈希值在查找值时需要用到。

正因如此，映射是没有长度的，也没有键集合或值集合的概念。

映射类型仅能用来作为状态变量，或者在内部函数中作为 storage 类型的引用。

可以通过将映射标记为 public，来让 Solidity 创建一个访问器。通过提供一个键值作为参数来访问它，将返回对应的值。

映射的值类型也可以是映射，在使用访问器访问时，要提供这个映射值所对应的键，不断重复这个过程。

下面是一段映射的示例代码：

```
pragma solidity ^0.4.20;

contract MappingExample {
    mapping(address => uint) public balances;
```

```
    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(this);
    }
}
```

8.1.14 包含左值的运算符

类似于 C 语言，在 Solidity 语言中可以运用左值运算符，例如，a+=e 等同于 a=a+e。同时其他的运算符如-=、*=、/=、%=、|=、&=和^=，它们的定义都类似。

例如，a++等同于 a+=1，a--等同于 a-=1，在其表达式中，a 保持原来的值；而在++a、--a 这样的表达式中，a 的值也发生了变化，存储是的+1、-1 后的 a 值。

8.1.15 删除

“delete a”的功能是对 a 这种变量类型赋初始值，例如，对于整数 a 的操作 delete a，等同于对 a 赋值 a=0。当 delete 应用于数组时，相当于赋值了一个长度为 0 的动态数组，或者一个所有成员都重置的相同长度的静态数组。当 delete 应用于结构时，相当于对结构的全部成员进行重置后的赋值。

delete 对映射不起作用，所以当我们删除（delete）一个结构时，它会重置所有不是映射的成员，并导入非映射的成员，但是其中的映射成员仍然会被删除。

因为 delete a 这条语句相当于对 a 重新赋值，所以其中存储的是一个新的变量。下面是一段示例程序，我们可以了解 delete 的使用。

```
pragma solidity ^0.4.20;
```

```
contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x;              // 设定 x 为 0，并不影响 data 变量
        delete data;           // 设定 data 为 0，并不影响仍持有副本的 x
        uint[] storage y = dataArray;
        delete dataArray;      // 设定了一个 0 长度的 dataArray，因为 uint[]是
一个复合变量
        // y 是一个存储对象的别名，会受到影响。另外，“delete y”无效，因为引用存
储对象赋值给本地变量只能来自已有的存储变量
    }
}
```

8.1.16　基本类型的转换

上面所说的各种变量类型都属于基本类型，如果要进行转换，则有以下两种方式。

1. 隐式转换

如果碰到在不同类型之间进行计算的运算符，则编译系统会进行隐式转换，将其中一种类型转换为另一种类型，这种情况在赋值时（将一种类型的值赋给另一种类型的变量）也会发生。通常情况下，只要语义合理，不发生信息丢失，不同类型之间的隐式转换是可以发生的，比如 uint8 转换为 uint16、int128 转换为 int256，但是 int8 不能转换为 uint256（因为 uint256 中没有负数）。此外，无符号整数可以转换为同样长度或更长的字节数组，但反过来就不行了。任何类型的变量，只要能转换为 uint160，就可以转换为 address（地址）变量。

2. 显式转换

如果编译器不允许隐式转换，并且开发人员对于转换规则和结果等比较了解，就可以采用显式转换，但有时可能会导致意想不到的结果，所以一般要在转换前先做一下测试，确保转换结果是自己想要的。下面是一个显式转换的例子：

```
int8 y = -3;
uint x = uint(y);
```

虽然 y 是 int8 类型的，不能直接转换成 uint 值，但是通过一个函数先将其转换成 uint 值，再赋值给 uint 的 x，就可以成立。

高位数据与低位数据之间在进行显式转换时，要对高位数据进行截断，如下面的例子：

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b 的值为 0x5678
```

a 为一个 32 位的无符号整数，要转换为无符号的 16 位整数，就要将低 4 位截断，所以 b 的值就是 0x5678。

在 Solidity 0.5.0 版本中，只允许长度相同的整数与固定长度的字节数组之间进行显式转换，所以在长度不同的整数与固定长度的字节数组之间进行转换时，先要将双方的长度调整到匹配，这就会涉及补齐和填充操作。例如以下程序：

```
uint16 x = 0xffff;
bytes32(uint256(x));    // 左侧补空
bytes32(bytes2(x));     // 右侧补空
```

8.1.17 类型推导

在变量类型未能明示的情况下，出于方便考虑，编译器会根据变量被赋值的第一个表达式的类型来推导出该变量的类型，例如：

```
uint24 x = 0x123;
var y = x;
```

其中，y 的类型根据 x 的第一次赋值确定为 uint24，虽然第二行中使用了 var 来对 y 进行定性，但是 var 不能用于函数参数或返回值，所以 y 的类型依然是 uint24。

由于类型推导的存在，所以有时需要注意对变量类型做好声明，避免类型推导所带来的溢出。例如，在循环语句“for (var i = 0; i < 2000; i++) {…}”中，变量 i 由类型推导确定为 uint8 类型，但完成循环之后其最大值为 2000，而 2000 已经超出了 uint8 类型所能表示的最大值范围（因为 2**8=256）。为了避免这种情况的发生，最好在一开始就对变量类型进行声明。

8.2　Solidity 语言的表达式和控制结构

本节我们继续了解采用 Solidity 语言进行以太坊应用开发的表达式和控制结构。

8.2.1　函数的输入参数和输出参数

与 JavaScript 语言相似，Solidity 语言的函数必须有输入参数；但是与 JavaScript 语言和 C 语言等不同的是，Solidity 语言的输出可以是任意数量的参数。

输入参数的声明方式与变量的声明方式相同。例如，假定编制的合约可以接收带有两个参数的外部调用，那么可以写成如下代码：

```
pragma solidity ^0.4.16;

contract Simple {
    function taker(uint _a, uint _b) public pure {
        …
    }
}
```

输出参数的声明与 returns 关键字之后的声明内容保持相同的语法。例如，在合约中希望输出给定的两个整数的和与积，则代码可以这样写：

```
pragma solidity ^0.4.16;

contract Simple {
    function arithmetics(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

输出参数的名称可以省略，输出值可以用 return 语句来指定。返回参数的

初始值为 0，如果不对其进行明确设定，它就持续保持为 0。

一般情况下，输入参数和输出参数都是在函数体中用作表达式的，但是它们也可以用在赋值时的等式左侧。

8.2.2 控制结构

JavaScript 语言中的大部分控制结构，除 switch 和 goto 外，都可以在 Solidity 语言中应用，包括 if、else、while、do、for、break、continue、return，以及条件判断“?:”等，而且其应用语法与 C 语言和 JavaScript 语言相同。

当用于条件句式时，括号不能省略，但是括在单一语句代码外面的花括号是可以省略的。

与 C 语言和 JavaScript 语言不同的是，Solidity 语言中没有布尔类型与非布尔类型的值之间的转换，因此“if (1) {…}”这样的句式在 Solidity 语言中是无效的。

8.2.3 返回多元值

如果一个函数有多个参数，如“return(v0,v1,…,vn)”，则可以返回多个数值。当然，return 数组的元素个数必须与函数输出参数的个数相符。

8.2.4 函数调用

函数调用包括内部函数调用与外部函数调用两种情况。

（1）内部函数调用：在本合约中的函数可以直接调用，或者递归调用，比如下面这个纯粹显示内部调用和递归调用的无意义的例子：

```
pragma solidity ^0.4.16;

contract C {
    function g(uint a) public pure returns (uint ret) { return f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

仅供非商业用途或交流学习使用

只有在同一个合约中的函数才能进行内部调用。在编译时函数调用被编译为对 EVM 的简单跳转，这样当前内存并不清除，相当于向内部调用函数有效地传递了一个内存引用。

（2）外部函数调用的常见形式为 this.g(8);及 c.g(2)（此处 c 为一个合约的名称），它需要借助信息调用，不能直接跳转。

要想使用在其他合约中的函数，必须采用外部函数调用。在执行外部函数调用时，函数所有的参数都要被复制到内存中。

此外，在调用外部函数时需要支付费用，通常采用 wei（伟）来计价，伴随调用而支付的 wei 的数量，以及支付的总的 Gas 费用情况，用两个特殊选项“.value()”和“.gas()”来分别确定。下面是外部函数调用的程序示例：

```
pragma solidity ^0.4.20;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) public { feed = InfoFeed(addr); }
    function callFeed() public { feed.info.value(10).gas(800)(); }
}
```

注意：在上面的程序中，修饰符 payable 必须写上，否则.value()就不可用了。

TIPS:

在执行外部函数调用时，如果被调用函数不存在，或者被调用函数自己释放出了异常，或者仅仅是调用者自己的 Gas 耗尽了，则都会导致外部函数调用出现异常。

8.2.5　函数参数的显名调用

在执行函数调用时，其参数的调用可以通过给定的参数名来实现，外部用花括号“{}”封装。在采用这种方式调用函数时，函数参数的名称一定要与函数声明中的参数名称相符，但先后顺序可以不一致，比如下面的例子：

仅供非商业用途或交流学习使用

```
pragma solidity ^0.4.20;

contract C {
    function f(uint key, uint value) public {
        // ...
    }

    function g() public {
        // 通过参数名称进行调用
        f({value: 2, key: 3});
    }
}
```

8.2.6 省略函数参数的名称

未使用的函数参数（特别是返回参数）的名称可以省略，这些参数当然还是存放于堆栈中的，但是不可访问，例如：

```
pragma solidity ^0.4.16;

contract C {
    // 省略了函数参数名称
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

8.2.7 在合约中创建新合约

使用 new 关键字可以在一个合约中再创建出另一个合约来，但是要求被创建的合约的代码要提前已经明确，所以不能使用递归式创建。在下面的例子中通过 new 关键字创建了新的合约。

```
pragma solidity ^0.4.20;

contract D {
    uint x;
    function D(uint a) public payable {
        x = a;
    }
```

仅供非商业用途或交流学习使用

```
}

contract C {
    D d = new D(4); //将作为 C 的构造器的组成部分运行

    function createD(uint arg) public {
        D newD = new D(arg);
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        //创建 newD 合约并发送以太币
        D newD = (new D).value(amount)(arg);
    }
}
```

在示例中，在创建 D 类型合约的例子（newD）时使用了“.value”选项，可以进行以太币的转账，但不能限定 Gas 费用的数量。如果创建失败（比如因堆栈溢出、账户余额不足等），则将触发异常。

8.2.8　解构赋值和返回多元值

在 Solidity 语言内部允许使用“元组（tuple）”结构。元组是一种对象列表，它的元素类型可以不同，但在编译时长度固定。元组可以用来一次性返回多元值，也可以用来给新声明的变量赋值，或者给已经存在的变量赋值。下面是一个示例程序：

```
pragma solidity >0.4.23 <0.5.0;

contract C {
    uint[] data;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // 用类型、返回元组赋值来声明变量
        (uint x, bool b, uint y) = f();
        // 赋值互换的通用做法，但不适用于非数值存储类型
        (x, y) = (y, x);
```

仅供非商业用途或交流学习使用

```
        // 组件可被忽略
        (data.length,,) = f(); // 设定长度为 7
        // 只有在赋值左侧的组件才可被忽略，产生一个异常
        (x,) = (1,);
        // (1,)是界定单元素元组的唯一方式。如果用(1)，则等于 1
    }
}
```

在上面的例子中，可以看到，如果元组中只有一个元素，则其表示方式是(1,)，其中的逗号不能省略；一旦省略，“(1)”等于 1，就不是元组了。

8.2.9 范围和声明

合约中声明的变量都会有一个初始默认值，这个默认值的各个位表示符可以全部为 0，无论变量是哪种类型，其“默认值”都是典型的“0 状态”。比如，布尔型变量 false 的默认值，以及 uint 或 int 类型的默认值都是 0。对于静态长度数组，从 bytes1 到 bytes32，它的每个独立元素都会初始化为与其类型对应的默认值。对于动态长度的数组，即 bytes 和 string，其默认值是一个空数组或空字符串。

函数内部任何位置的变量声明在该函数的全部范围内都是有效的，不管这个声明是在函数内的前部还是后部进行的，这是因为 Solidity 语言继承了 JavaScript 语言的“范围规则”。与此相对应的是，在其他一些语言中，变量声明的有效性范围是从声明位置起到函数结束的代码段范围。

由于这一规则的存在，下面的程序代码就会出错：

```
pragma solidity ^0.4.16;

contract ScopingErrors {
    function scoping() public {
        uint i = 0;

        while (i++ < 1) {
            uint same1 = 0;
        }

        while (i++ < 2) {
            uint same1 = 0;// 非法，这是 same1 的第二次声明
```

仅供非商业用途或交流学习使用

```
        }
    }

    function minimalScoping() public {
        {
            uint same2 = 0;
        }

        {
            uint same2 = 0;// 非法，这是 same2 的第二次声明
        }
    }

    function forLoopScoping() public {
        for (uint same3 = 0; same3 < 1; same3++) {
        }

        // 非法，这是 same3 的第二次声明
        for (uint same3 = 0; same3 < 1; same3++) {
        }
    }
}
```

8.2.10　错误处理：断言、请求、还原与异常

Solidity 语言采用“状态还原”异常处理机制来控制错误，这种异常处理机制能够恢复当前调用（及其所有子调用）对于状态的改变情况，并给调用者标出错误。可以使用“便利函数”——assert()及 require()检查各种条件，并在条件不符合时释放出异常。assert()函数只能用于测试内部错误和不变量；而 require()函数用来确保有效条件（如输入或合约状态变量）保持符合，或者对从外部合约调用的返回值进行验证。适当的函数代码一定不会产生无效的 assert 状态，所以，如果这种情况发生了，就说明程序中存在 Bug，需要开发人员进行查找和修复。

另外，还有两种方式可以触发异常：revert()函数可以标记错误并恢复当前调用的状态，它会给调用者传回一个包含错误细节的字符串信息；另一个函数 throw()也可以实现 revert()函数的功能，但不能返回字符串信息，而且在 Solidity 0.4.13 版本以后，throw()函数已经被废弃了。

仅供非商业用途或交流学习使用

在下面的例子中，使用 require()函数进行输入条件的检查，使用 assert()函数进行内部错误的检查。

```
pragma solidity ^0.4.22;

contract Sharer {
    function sendHalf(address addr) public payable returns (uint
balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = this.balance;
        addr.transfer(msg.value / 2);
        // 由于转账释放一个错误异常且在此处不能回滚，所以仍然持有一半资金
        assert(this.balance == balanceBeforeTransfer - msg.value / 2);
        return this.balance;
    }
}
```

在下列情况下，通常产生 assert 类型的异常。

（1）在访问数组时，索引值超过数组长度或索引为负，即用 x[i]访问数组 x，当 i>x.length 或 i<0 时。

（2）在访问固定长度的字符数组 bytesN 时，索引值过大或为负。

（3）除数为 0 或对 0 求模（如 6/0 或 6%0）。

（4）对负数进行移位操作。

（5）把一个过大或为负的值转换为枚举类型。

（6）在内部函数中调用一个初始值为 0 的变量。

（7）调用 assert()函数时的参数，其值被赋为 false。

在下列情况下，通常产生 require 类型的异常。

（1）调用 throw()函数。

（2）调用 require()函数时的参数，其值被赋为 false。

（3）在通过信息调用来调用某个函数时，这个函数未能正常完成（比如，Gas 费用耗光，或者函数类型不匹配，或者它自身触发了异常）。不过，在使用低阶操作如 call、send、delegatecall 或 callcode 时不会触发 require 类型的异常，它们只返回 false。

仅供非商业用途或交流学习使用

（4）用 new 关键字创建了一个新的合约，但又没有正常完成。

（5）调用的外部函数中并无代码。

（6）合约通过一个公共函数来接收以太币，而这个公共函数却没有 payable 修饰语。

（7）合约通过公共取值函数来接收以太币。

（8）.transfer 转账失败。

在内部，Solidity 语言对于 require 类型的异常执行“还原”操作，而对 assert 类型的异常执行“无效”操作。这两种情况都会导致以太坊虚拟机对状态的改变进行复原，因为期望的效果没达到，导致没有安全的方式去继续执行代码。如果我们想保留交易的基本特性，那么最安全的事就是还原所有的改变，并让整个交易（至少是调用）无效。

下面的例子展示了错误信息字符串与 require 异常和还原操作共同生效的情况。

```
pragma solidity ^0.4.22;

contract VendingMachine {
    function buy(uint amount) payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // 替代做法
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // 完成购买
    }
}
```

这里所提供的错误信息字符串是二进制接口编码的字符串，上例中的语句“revert("Not enough Ether provided.");”将产生如下的十六进制代码作为返回的错误信息。

```
0x08c379a0   //用于 Error(string)的函数选择器
0x0000000000000000000000000000000000000000000000000000000000000020
// 数据偏移
```

仅供非商业用途或交流学习使用

```
0x000000000000000000000000000000000000000000000000000000000000001a
// 字符串长度
0x4e6f7420656e6f756768204574686572207072f76696465642e000000000000
// 字符串数据
```

8.3 以太坊合约

以太坊应用程序的主要结构就是合约（Contract），合约中包含着一些用状态变量表示的持续性变量，以及可以对这些变量进行修改的函数。

8.3.1 创建合约

创建合约有两种方式：一是通过以太坊交易从外部创建；二是在 Solidity 合约的内部创建。创建过程可以使用各种集成开发环境，如 Remix，它无缝集成了用户界面元素，让创建合约的效率大为提高。在编程方面，创建合约最好的做法是应用 JavaScript 应用程序接口 web3.js，通过应用 web3.eth.Contract 方法，可以加快合约的创建过程。

在创建合约时，它的构造器（采用 constructor 关键字声明的一个函数）执行一次。构造器是可选的，但一个合约只允许采用一个构造器，不允许超载。在合约代码之后，构造器的参数应该以二进制接口代码的形式传递。如果开发人员采用的是 web3.js 编程，那么对此无须过多关注，因为 web3.js 会进行处理。

如果一个合约要创建另一个合约，那么创建者需要全部了解被创建者的代码，不允许创建循环依赖关系。

下面是一个创建合约的示例：

```
pragma solidity ^0.4.22;

contract OwnedToken {
    // 代币创建器是下面定义的一种合约类型，当它未使用时可被用来创建新合约
    TokenCreator creator;
    address owner;
    bytes32 name;
```

仅供非商业用途或交流学习使用

```
    // 这是一个构造器，用于创建器注册和名称赋值
    constructor(bytes32 _name) public {
        // 状态变量可以通过名称访问，但不能用 this.owner 方式访问。这也适用于函数，
尤其是在构造器中，只能在内部调用它们，因为合约本身尚不存在
        owner = msg.sender;
        // 进行明确的类型转换，从“address”转换为“TokenCreator”，假定调用合
约为 TokenCreator，此时并无实际途径来进行检验
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // 只有创建者才可以更改合约名称。由于合约隐式转换为地址，所以需要进行合约
调用者地址与创建者地址的比较

        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // 只有当前持有者可以转让代币
        if (msg.sender != owner) return;
        // 这里要调用下面定义的合约函数，如果调用失败（如 Gas 耗尽），则马上停止执行
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name)
       public
       returns (OwnedToken tokenAddress)
    {
        // 创建一个新的代币合约并返回其地址。在 JavaScript 语言中，这种返回类型是
单一的“address”，与二进制程序接口类型很接近
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name)  public
{
        // 外部类型的“tokenAddress”是简单的“address（地址）”类型
        tokenAddress.changeName(name);
    }
```

```
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        view
        returns (bool ok)
    {
        // 检查一些专门条件
        address tokenAddress = msg.sender;
        return (keccak256(newOwner) & 0xff) == (bytes20(tokenAddress) &
0xff);
    }
}
```

8.3.2 可见性

在 Solidity 语言中有两种库函数调用方式：内部调用（也可以称为消息调用）和外部调用，因此函数和状态变量的可见性分为 4 种类型。

函数明确规定有 4 种类型：外部（external）、公共（public）、内部（internal）和私有（private），默认类型为 public。对于状态变量，除 external 类型不可用之外，其他 3 种类型均存在，默认类型为 internal。

4 种类型的具体情况如下：

（1）external。外部函数作为合约接口的组成部分，可以采用交易的方式在合约外部对它们进行调用。假设有一个外部函数 f()，它不能在合约内部被调用，也就是说，f()的方式不能运行，但 this.f()这种方式可以运行。在接收大批量数据的数组时，采用外部函数更有效。

（2）public。公共函数是合约接口的组成部分，它既可以在合约内部被调用，也可以通过消息来调用。如果一个状态变量是公共的，则会产生一个自动取值函数。

（3）internal。这种类型的函数和状态变量只能在合约内部被访问，即当前合约及当前合约的派生合约，外部调用的方式如 this.f()是不可用的。

（4）private。私有类型的函数和状态变量只在定义它们的合约中是可见的，即使在派生合约中也不可见。

可见性说明符在语句中位于状态变量的类型之后，在函数声明时位于参数列表与返回参数列表之间，例如：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

在下面的示例代码中，合约 D 可以调用合约 C 的 c.getdata()函数从存储器中取得 data 变量的值，但合约 D 不能调用合约 C 中的私有函数 f()；合约 E 是从合约 C 中派生出来的，因此合约 E 可以调用合约 C 的内部函数 compute()。

```
pragma solidity ^0.4.20;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return
a + b; }
}

contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7);        // 错误：成员函数 f()不可见
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); //错误：成员函数 compute()不可见
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); //访问内部成员（从父合约中导出）
    }
}
```

8.3.3 取值函数

对于所有的公共状态变量，编译器将自动为其创建取值函数（Getter Function）。在下面的示例程序中，编译器创建了一个调用状态变量 data 的函数，这个函数没有其他参数，只返回一个 uint 值，即状态变量 data 的数值。状态变量的初始化在声明过程中完成。

```
pragma solidity ^0.4.20;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public {
        uint local = c.data();
    }
}
```

取值函数有外部可见性。如果代码显示是内部访问（在变量名前面没有函数名“this.”），则将取值函数作为状态变量进行赋值；如果代码显示是外部访问（在变量名前面有函数名“this.”），则将取值函数作为函数进行赋值。例如：

```
pragma solidity ^0.4.20;

contract C {
    uint public data;
    function x() public {
        data = 3;                   // 内部访问
        uint val = this.data();     // 外部访问
    }
}
```

8.3.4 函数修饰符

函数修饰符可以很容易地改变函数的特性，比如，它们可以自动检查执行函数的前置条件。函数修饰符可以从合约的属性中继承，也可以被派生函数所修改。

下面是一段程序代码，其中应用了函数修饰符。

```
pragma solidity ^0.4.22;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;

    // 这个合约只定义了一个函数修饰符，但自己并不使用，会在派生合约中用到
    // 在下面的修饰符定义中，函数本体将在特殊符号 "_;" 处插入
    // 其含义是：如果 owner 调用这个函数，则函数得以执行；否则就触发异常

    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract mortal is owned {
    // 这个合约从 owned 合约处继承了 onlyOwner 修饰符，并应用于其函数 close()。
该函数的功能是：保证只有 owner 来调用 close()函数才有效
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // 修饰符可以接收参数传递
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    function Register(uint initialPrice) public { price = initialPrice; }
```

```
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// 下面的函数有互斥保护，在 msg.sender.call 中不能再次调用 f()函数
    /// "return 7"的含义是将 7 赋给返回值，但在修饰符中仍然执行"locked = false"
    function f() public noReentrancy returns (uint) {
        require(msg.sender.call());
        return 7;
    }
}
```

使用空格分隔的列表在一个函数中也可以使用多个修饰符，Solidity 会根据呈现顺序逐一赋值。

修饰符的参数可使用任意表达式。在这种情况下，函数中的全部可见符号对于修饰符来说都是可见的，但在修饰符中引入的符号在函数中则不可见。

8.3.5 常数状态变量

状态变量可以被声明为常数，此时它们必须赋值为一个表达式，这个表达式在编译时是一个常数。在这种情况下，访问存储器、区块链数据的任何表达式（如 now、this.balance 或者 block.number），或者执行数据（如 msg.value 或

gasleft()等），或者进行外部合约调用，都是不允许的。但构造函数是允许使用的，即便它们进行了外部调用，如 keccak256、sha256、ripemd160、ecrecover、addmod、mulmod 等构造函数。

当前并不是所有的变量类型都支持常数类型，仅有数值和字符串两种变量类型支持常数类型。例如：

```
pragma solidity ^0.4.20;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

8.3.6　视图函数

有时候，在函数的声明中包含 view（视图）关键字，这表示它们不会被修改状态。

例如：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
}
```

以下这些操作可被视为修改函数状态：

（1）写入状态变量。

（2）释放事件。

（3）创建其他合约。

（4）使用“selfdestruct（自毁）”函数。

（5）通过调用来传送以太币。

（6）调用未标识“view”或“pure”的任何函数。

（7）使用低阶调用。

（8）使用包含大量操作码的内联汇编（Inline Assembly）。

8.3.7 纯函数

当函数被声明为 pure（纯函数）时，表示它们既不能被读取，也不能被修改状态。

例如：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

在上文所说的 8 种修改函数状态的操作之外，还有一些操作可被视为对状态的读取，包括：

（1）读取状态变量.

（2）访问 this.balance 或<address>.balance。

（3）访问区块（block）、交易（Tx）、消息（Msg）的任意成员。

（4）调用未标识“pure”的任意函数。

（5）使用包含大量操作码的内联汇编。

8.3.8 后备函数

在合约中存在这样一种未命名的函数：它没有参数，也不返回任何内容，它只在一种情况下运行，即当调用该合约时，其他函数都不符合给定的函数标识符，或者根本没有提供数据。这种函数称为后备函数，它实质上是一种设计出来的退路。

另外，当合约收到无数据的以太币时也会执行后备函数。为了接收以太币，

后备函数必须被声明为 payable；如果没有后备函数，合约就不能通过常规交易接收以太币。

后备函数在极端情况下只实施登录操作，因此最少只需要 2300 的 Gas 费用，而其他的操作所消耗的 Gas 费用都远远高于 2300，如写入存储器，创建合约，调用消耗大量 Gas 费用的外部函数，发送以太币等。当然，只要 Gas 费用足够，后备函数与其他函数一样，也可以执行大量的复杂操作。

以下是后备函数的例子：

```
pragma solidity ^0.4.20;

// 这个合约保存所有发来的以太币，并且不能退回

contract Sink {
    function() public payable { }
}
```

8.3.9　重载函数

一个合约可以有多个名称相同但参数不同的函数，这种情况也适用于派生函数。在下面的例子中，在合约 A 的范围内对 f()函数进行了重载(Overloading)。

```
pragma solidity ^0.4.16;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }

    function f(uint _in, bytes32 _key) public pure returns (uint out)
{
        out = 2;
    }
}
```

重载函数也可应用于外部调用。

重载函数的选择是将当前范围内的函数声明情况与函数调用所提供的参数进行匹配来完成的，如果所有的参数都可以隐含地转换为期望的类型，那么所

有的函数都可以被选作重载候选函数；反之，如果没有一个函数能够匹配，那么方案失败。另外需要注意的是，在匹配重载函数的参数时，返回参数是不予考虑的。

例如：

```
pragma solidity ^0.4.16;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

在上面的例子中，如果调用 f(50)，则产生类型错误，因为 50 既可以转换为 uint8 类型，也可以转换为 uint256 类型，也就是说上面的两个函数都满足重载函数的要求。而调用 f(256)就不会出现这样的错误，因为 256 只能转换为 uint256 类型，而不能转换为 uint8 类型。

8.3.10 事件

“事件”机制的存在使得 Solidity 可以方便地使用以太坊虚拟机的日志工具，以此来“调用” JavaScript 的回调工具，在 DApp 的用户界面上监听各种事件。

事件是合约的继承成员，当事件被调用时，它会将参数存储在交易日志中。交易日志是区块链的一个特殊数据结构，这些日志与合约地址相关联，将被封装到区块链之中，只要区块可以访问，就一直保存在区块上。

在日志中可以应用简化付款验证（SPV）作为证明。如果一个外部对象提供了这样的证明，则可以用来检查日志是否真实存在于区块链中，这时必须提供区块头数据，因为合约只能看到最新的 256 个区块的哈希值。

当使用了定语“indexed”时，事件最多携带 3 个参数，在这种情况下每个参数都可以被搜索到，被索引（indexed）的参数在用户界面中也可以过滤掉特

定值。如果数组（包括字符串数组和字节数组）用作被索引的参数，那么它的 Keccak-256 哈希值将作为主题来存储（而不再作为日志来存储）。如果事件在声明时带上了 anonymous 指示符，那么事件签名的哈希值也是主题的内容之一，这时就不能通过事件名来过滤掉特定的匿名事件了。所有非索引的参数作为日志的组成部分来存储。

例如：

```
pragma solidity ^0.4.21;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // 使用 emit()函数来释放事件,包括事件名、括号中的参数。通过搜寻 Deposit()
函数，任何调用（甚至深度嵌套）都可以在 JavaScript 应用程序接口中被探测到
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

在 JavaScript 应用程序接口中，事件的应用过程如下例所示：

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var event = clientReceipt.Deposit();

// 监测变化
event.watch(function(error, result){
    // 结果（result）中会包含各种信息，包括用于 Deposit()函数调用的参数
    if (!error)
        console.log(result);
});

// 或传递一个回调信号，立即启动监听
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
```

仅供非商业用途或交流学习使用


```
        console.log(result);
});
```

8.3.11 继承

Solidity 支持多重继承，具体方式是复制代码及其包括的多样性。当一个合约是从多个合约（称之为基础合约）继承而来的时，在区块链上只创建了一个单一的合约，所有基础合约中的代码都被复制到这个创建的合约中。

下面是一个关于合约继承的例子：

```
pragma solidity ^0.4.22;

contract owned {
    constructor() { owner = msg.sender; }
    address owner;
}

// 下面使用 is 来派生出另一个合约。派生的合约可以访问所有非私有成员（包括内部函数
和状态变量），但这些数据不能借助 this 来进行外部访问
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

// 提供这些抽象合约是为了构建编译器可接受的接口。注意函数没有函数体。如果一个合约
不执行所有函数，那么它只能被用作接口
contract Config {
    function lookup(uint id) public returns (address adr);
}

contract NameReg {
    function register(bytes32 name) public;
    function unregister() public;
 }

// 多重继承是合法的。注意 owned 也是 mortal 的基类
contract named is owned, mortal {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2
```

仅供非商业用途或交流学习使用


```
f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // 通过使用另一个同名函数和相同的输入数据或类型，原函数可被覆写。但如果覆写函
数的输出参数类型不同，就会出错。在调用本地函数或消息函数时都可以考虑覆写这种方法
    function kill() public {
        if (msg.sender == owner) {
            Config  config  =  Config(0xD5f9D8D94886E70b06E474c3fB14Fd
43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // 仍然可以调用一个具体的覆写函数
            mortal.kill();
        }
    }
}

contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) public {
     if (msg.sender == owner) info = newInfo;
   }

   function get() public view returns(uint r) { return info; }

   uint info;
}
```

在上面的例子中，我们调用了 mortal.kill()函数来传递“解构”需求，但是这种方式存在一些问题，比如下面的例子：

```
pragma solidity ^0.4.22;

contract owned {
    constructor() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

仅供非商业用途或交流学习使用


```
contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ mortal.kill(); }
}

contract Final is Base1, Base2 {
}
```

在调用 final.kill()函数时会调用 Base2.kill()函数作为直接的派生函数进行覆写，但这时就会绕开 Base1.kill()函数，因为它甚至不知道有 Base1。解决方法是使用 super，代码如下：

```
pragma solidity ^0.4.22;

contract owned {
    constructor() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ super.kill(); }
}

contract Final is Base1, Base2 {
}
```

仅供非商业用途或交流学习使用

8.3.12　构造器

构造器是采用 constructor 进行声明的一种可选函数，它用于合约创建的执行。构造器函数可以是两种类型之一：public 和 internal。如果没有指定构造器函数，那么合约会采用默认的构造器“constructor() public {}”。代码如下：

```
pragma solidity ^0.4.22;

contract A {
    uint public a;

    constructor(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    constructor() public {}
}
```

构造器如果被设定为 internal，则将使合约成为抽象合约（下文会讲到）。

所有基础合约的构造器都要调用线性规则。如果基础合约的构造器有参数，则派生合约需要确定全部参数，可以采用如下所示的两种方式：

```
pragma solidity ^0.4.22;

contract Base {
    uint x;
    constructor(uint _x) public { x = _x; }
}

contract Derived1 is Base(7) {
    constructor(uint _y) public {}
}

contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) public {}
}
```

一种方式是从继承列表中直接继承（is Base(7)）；另一种方式是在派生的构造器的头部内容中调用修饰符作为组成部分（Base(_y * _y)）。如果构造器参数是常量，那么采用第一种方式更方便。如果基础合约的构造器参数依赖派生

仅供非商业用途或交流学习使用

合约的构造器参数，则采用第二种方式。构造器参数要在继承列表中给出，或者在派生合约的构造器的修饰符类型中给出，而在这两个地方对构造器函数进行定义可能会导致出错。如果一个派生合约不能定义它所有的基础合约的构造器参数，那么它就是一个抽象合约。

8.3.13 抽象合约

当合约中至少有一个函数缺乏执行代码时，这个合约就被标注为抽象合约。例如：

```
pragma solidity ^0.4.20;

contract Feline {
    function utterance() public returns (bytes32);
}
```

这样的合约不能被编译。有一些这样的合约，其中可执行代码与不可执行代码共存，这样的合约也不能被编译。但是它们可以作为派生合约的基础合约，例如：

```
pragma solidity ^0.4.20;

contract Feline {
    function utterance() public returns (bytes32);
}

contract Cat is Feline {
    function utterance() public returns (bytes32) { return "miaow"; }
}
```

如果一个合约是从抽象合约派生而来的，并且不能通过覆写来执行所有的非执行函数，那么这个派生的合约本身也是一个抽象合约。

这里需要注意的是，不具备执行性的函数与普通的函数类型在句法上很相似，但二者其实是不同的。下面是一个不具备执行性的函数声明：

```
function foo(address) external returns (address);
```

而正常的函数类型声明是这样的：

```
function(address) external returns (address) foo;
```

仅供非商业用途或交流学习使用

8.3.14　接口

接口（Interface）与抽象合约相似，但是它们没有任何可执行函数，此外还有一些进一步的限制，如下：

（1）不能继承其他合约或接口。

（2）不能定义构造器。

（3）不能定义变量。

（4）不能定义结构。

（5）不能定义枚举。

接口受到合约二进制接口表示范围的限制，它与二进制接口之间的转换是被允许的，而且不会丢失任何信息。

接口以其专用关键字来表示，如下：

```
pragma solidity ^0.4.11;

interface Token {
    function transfer(address recipient, uint amount) public;
}
```

合约在继承过程中也可以从其他合约处将接口继承过来。

8.3.15　库

库与合约相似，但库的特点是只在特定地址部署一次，其代码通过调用函数 delegatecall()可以重复使用。这意味着当调用库函数时，其代码是在调用合约中运行的，即 this 指向调用函数，而且调用函数的存储器也可以被访问。由于库函数是源代码中的独立部分，所以它只能访问调用函数的状态变量，而且需要提供这些变量的详情。

库函数如果没有修改状态（当它们是 view 或 pure 状态时），就只能直接调用（不使用 delegatecall()），因为库函数的设定状态是非静态的。尤其需要注意的是，库函数是不能被销毁的，除非在采用某些方法规避 Solidity 关于类型的系统规则后才能实现。

仅供非商业用途或交流学习使用

在使用库函数的合约中，库函数被视为该合约的隐含基础合约；在继承架构下，库函数不是显性可见的，但是调用库函数的方式与调用显性基础合约的方式一样，即：如果 L 是一个库函数的名称，则可以用“L.f()”方式来调用库函数。

此外，库函数中声明为 internal 的函数在所有的合约中都可见，这使得库函数看上去就像基础合约一样。在调用内部函数时使用的是内部调用规则，即所有的内部类型变量和内存类型变量都可以通过引用来传递，而不用复制。要在以太坊虚拟机中实现这一点，内部库函数的所有代码，以及所有当时调用的函数，在编译时都要被拉进调用合约，此时使用常规函数 jump()代替低阶函数 delegatecall()。

下面的例子展示了如何使用库函数。

```
pragma solidity ^0.4.22;

library Set {
  // 下面定义一个新的结构类型，用来保存调用合约中的数据
  struct Data { mapping(uint => bool) flags; }

  // 注意它的第一个参数是关于类型引用的“storage reference”，因此只将它的存储地址而非内容作为调用的组成部分进行传递。如果该函数可看作对象的方法，则习惯做法是调用第一个参数“self”
  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // 已经存在
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // 未存在
      self.flags[value] = false;
```

仅供非商业用途或交流学习使用


```
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    Set.Data knownValues;

    function register(uint value) public {
        // 库函数无须指定该库函数的特定实例，因为“实例”就是当前合约
        require(Set.insert(knownValues, value));
    }
}
```

当然，不定义结构数据类型也可以使用库函数，没有任何存储引用参数，或者在任何位置都存在多重存储引用参数，库函数都可以正常使用。

下面的例子展示了怎样利用库函数所具有的内存类型、内部函数来执行定制化的类型，而不需要使用外部函数调用。

```
pragma solidity ^0.4.16;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    function fromUint(uint x) internal pure returns (bigint r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint _a, bigint _b) internal pure returns (bigint r)
{
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
        uint carry = 0;
```

仅供非商业用途或交流学习使用


```
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && carry > 0))
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // 增加一个分支
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal pure returns (uint)
{
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for BigInt.bigint;

    function f() public pure {
        var x = BigInt.fromUint(7);
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

在 Solidity 语言的当前版本中，库与合约的区别是：没有状态变量，不能继承或被继承，不能接收以太币。

仅供非商业用途或交流学习使用

8.3.16　用于“using for”

指令“using A for B”可以把来自函数库 A 中的库函数附加到任何类型 B 中，这些函数会接收它们调用的对象，并将其作为函数的第一个参数。而指令“using A for *”的作用是把函数库 A 中的库函数附加给任意类型。在这两种情形下，所有的函数，即使第一个参数的类型与目标类型不匹配的函数也会附加过去。在这个函数被调用的时点再进行类型检验，并完成函数覆写。

指令“using A for B”在当前范围内有效，限定在当前合约内。当使用了“using for”句式后，我们可以对上文的库函数程序进行重新编写，最终代码如下：

```
pragma solidity ^0.4.16;

library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
        return false; // 已存在
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // 未存在
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
  {
```

```
        return self.flags[value];
    }
}

contract C {
    using Set for Set.Data; // 此处为关键变化
    Set.Data knownValues;

    function register(uint value) public {
        // 在这里，Set.Data 类型的所有变量已经回应了成员函数。下面的函数调用等同
于 Set.insert(knownValues, value)
        require(knownValues.insert(value));
    }
}
```

需要注意的是，所有的库函数调用实际上都是 EVM 函数调用，因此，在传递内存类型或数值类型时，采用的是复制的方式，包括 self 变量在内也是复制完成的，只有在使用存储器引用变量时才不使用复制方式。

8.4 Solidity 汇编

Solidity 定义了一种汇编语言，在非 Solidity 环境下也可以使用。这种汇编语言也可以作为“内联汇编”在 Solidity 语言源代码中使用。下面分别介绍内联汇编和独立汇编，使读者对 Solidity 汇编有一个明确的认识。

8.4.1 内联汇编

为了实现更细致的控制，特别是为了通过编写函数库来提升编程语言的性能，Solidity 语言允许使用以下这种方式：在 Solidity 语句中插入内联汇编语句，这种语句是一种接近于虚拟机的语言。因为 EVM（以太坊虚拟机）实际上是一种堆栈机制，经常难以表述正确的堆栈区，也难以给位于堆栈正确位置的操作码提供参数，所以 Solidity 的内联汇编试图解决在编写汇编语句时遇到的这些问题及其他问题，它的特点如下。

（1）用函数类型的操作码替代传统表述，如用 mul(1, add(2, 3))来替代 push1

3 push1 2 add push1 1 mul。

（2）应用了本地汇编的变量，例如：

```
let x := add(2, 3)  let y := mload(0x40)  x := add(x, y)。
```

（3）可访问外部变量，例如：

```
function f(uint x) public { assembly { x := sub(x, 1) } }
```

（4）可使用标签，例如：

```
let x := 10  repeat: x := sub(x, 1) jumpi(repeat, eq(x, 0))
```

（5）可使用循环，例如：

```
for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }
```

（6）可使用 if 句式，例如：

```
if slt(x, 0) { x := sub(0, x) }
```

（7）可使用 switch 句式，例如：

```
switch x case 0 { y := mul(x, 2) } default { y := 0 }
```

（8）可使用函数调用，例如：

```
function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x,
f(sub(x, 1))) }   }
```

在下面的例子中提供了一个库函数，它要访问另一个合约的代码，并把这个合约写入 bytes 变量中。在纯粹的 Solidity 语言中这些功能是没法实现的，解决方法是采用汇编库函数，以增强 Solidity 语言的功能。代码如下：

```
pragma solidity ^0.4.20;

library GetCode {
    function at(address _addr) public view returns (bytes o_code) {
        assembly {
            // 需要使用汇编语言来取得代码的大小
            let size := extcodesize(_addr)
            // 配置输出的字节数组。这一功能在非汇编环境下也可以用“o_code = new
bytes(size)”来完成
            o_code := mload(0x40)
            // 包括补位数据的新“存储端”
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f),
not(0x1f))))
```

```
            // 在内存中存储长度
            mstore(o_code, size)
            // 需要使用汇编语言来取得实际代码
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

在优化器不能产生有效率的代码时，内联汇编也可以发挥有益的作用。

8.4.2 句法

和 Solidity 语言完全相同，汇编语言也能解析注释、常量、标识符，所以我们可以采用常见的“//”或“/*……*/”进行注释。内联汇编用“assembly {……}”来标识，在这个表达式的花括号中可以应用以下内容：

（1）常量，如 0x123、42 或"abc"（不超过 32 位字符的字符串）。

（2）操作码，如 mload、sload、dup1、sstore。

（3）以函数形式出现的操作码，如 add(1, mlod(0))。

（4）标签，如“name:”。

（5）变量声明，如 let x := 7, let x := add(y, 3)或者 let x（其中 x 的初始值被赋为 0）。

（6）标识符（包括标签、本地汇编变量、内联汇编使用的外部变量等），如 jump(name)、3 x add。

（7）赋值，如 3 =: x。

（8）以函数形式进行赋值，如 x := add(y, 3)。

（9）局部变量的作用域位于其中的代码块，如{ let x := 3 { let y := add(x, 1) }。

8.4.3 操作码

操作码如果带有参数，它们在形式上就会带有一个大括号，参数的顺序是可见的，也可以转换为非函数类型。用“-”标识的操作码不能把数据压入堆栈

仅供非商业用途或交流学习使用

中，用“*”标识的是另一种特殊类型，其他所有的操作码都能够向堆栈中压入数据。

“mem[a……b]”用来识别内存中从位置 a 开始直到位置 b（不包括 b）的字节；“storage[p]”用来识别位置 p 的存储内容。操作码“pushi”和“jumpdest”不能直接使用。

如表 8.5 所示为各种操作码指令的具体含义。

表 8.5　各种操作码指令的具体含义

指　令	含　义
stop	停止执行。效果等同于 return(0,0)
add(x, y)	x + y
sub(x, y)	x - y
mul(x, y)	x * y
div(x, y)	x / y
sdiv(x, y)	x / y，用于有符号的二进制补码数据
mod(x, y)	x % y
smod(x, y)	x % y，用于有符号的二进制补码数据
exp(x, y)	x 的 y 次方
not(x)	~x，对 x 按位求非
lt(x, y)	如果 x<y，则返回 1；否则返回 0
gt(x, y)	如果 x>y，则返回 1；否则返回 0
slt(x, y)	如果 x<y，则返回 1；否则返回 0，用于有符号的二进制补码数据
sgt(x, y)	如果 x>y，则返回 1；否则返回 0，用于有符号的二进制补码数据
eq(x, y)	如果 x 等于 y，则返回 1；否则返回 0
iszero(x)	如果 x 等于 0，则返回 1；否则返回 0
and(x, y)	对 x 和 y 按位求与
or(x, y)	对 x 和 y 按位求或
xor(x, y)	对 x 和 y 按位求异或
byte(n, x)	返回 x 中第 n 位的字节（其中最重要的是第 0 位的字节）
shl(x, y)	y 向左逻辑移位 x 位
shr(x, y)	y 向右逻辑移位 x 位
sar(x, y)	y 向右算术移位 x 位
addmod(x, y, m)	(x + y) % m 任意算术精度

仅供非商业用途或交流学习使用

续表

指　令	含　义
mulmod(x, y, m)	(x * y) % m 任意算术精度
signextend(i, x)	对 x 从第(i*8+7)位开始进行符号扩展
keccak256(p, n)	keccak(mem[p…(p+n)))
jump(label)	跳转到标签或代码位置
jumpi(label, cond)	当条件变量 cond 为非零时，跳转到标签处
pc	代码的当前位置
pop(x)	移除由 x 压入的元素
dup1 … dup16	复制第 i 个堆栈到顶部（从顶部开始计算）
swap1 … swap16	对第 i 个堆栈及顶部堆栈进行互换
mload(p)	mem[p..(p+32))
mstore(p, v)	mem[p..(p+32)) := v
mstore8(p, v)	mem[p] := v & 0xff（只修改单个字节）
sload(p)	storage[p]
sstore(p, v)	storage[p] := v
msize	内存大小，即最大可访问内存的索引
gas	现存可用于执行的 Gas 费用
address	当前合约或执行代码的地址
balance(a)	地址 a 的账户余额（以 wei 为单位）
caller	调用的发送器（不包括 delegatecall）
callvalue	与当前调用同步发送的 wei 值
calldataload(p)	调用从位置 p（32 字节）开始的数据
calldatasize	以字节为单位的调用数据的大小
calldatacopy(t, f, s)	从位于 f 位置的调用数据中复制 s 字节到内存的 t 位置
codesize	当前合约或执行部件的代码大小
codecopy(t, f, s)	从位于 f 位置的代码中复制 s 字节到内存的 t 位置
extcodesize(a)	地址 a 的代码大小
extcodecopy(a, t, f, s)	与 codecopy(t, f, s)相似，但复制的是在地址 a 处的代码
returndatasize	最后返回数据的大小
returndatacopy(t, f, s)	从位于 f 位置的返回数据中复制 s 字节到内存的 t 位置
create(v, p, s)	用代码 mem[p..(p+s))来创建新合约，发送 v 数量的 wei，返回新地址
create2(v, n, p, s)	用位于地址 keccak256(<address> . n . keccak256(mem[p..(p+s)))处的代码 mem[p..(p+s))来创建新合约，发送 v 数量的 wei，返回新地址

仅供非商业用途或交流学习使用

续表

指　令	含　义
call(g, a, v, in, insize, out, outsize)	用 mem[in..(in+insize))的输入来调用位于地址 a 处的合约，提供 g 个 Gas 和 v 数量的 wei，输出的区域位于 mem[out..(out+outsize))，错误时返回 0，成功时返回 1
callcode(g, a, v, in, insize, out, outsize)	等同于 call，但只使用来自 a 的代码并保持在当前合约范围内
delegatecall(g, a, in, insize, out, outsize)	等同于 callcode，但保持了 caller()和 callvalue()函数
staticcall(g, a, in, insize, out, outsize)	等同于 call(g, a, 0, in, insize, out, outsize)，但不允许状态修改
return(p, s)	结束执行，返回数据 mem[p..(p+s))
revert(p, s)	结束执行，恢复改变的状态，返回数据 mem[p..(p+s))
selfdestruct(a)	结束执行，注销当前合约并发送资金到 a
invalid	以无效指令来中止执行
log0(p, s)	无主题进行对数计算，数据为 mem[p..(p+s))
log1(p, s, t1)	以主题 t1 进行对数计算，数据为 mem[p..(p+s))
log2(p, s, t1, t2)	以主题 t1、t2 进行对数计算，数据为 mem[p..(p+s))
log3(p, s, t1, t2, t3)	以主题 t1、t2、t3 进行对数计算，数据为 mem[p..(p+s))
log4(p, s, t1, t2, t3, t4)	以主题 t1、t2、t3、t4 进行对数计算，数据为 mem[p..(p+s))
origin	交易发送器
gasprice	交易的 Gas 价格
blockhash(b)	区块号码为 b 的哈希值，只用于不含当前区块的最新 256 个区块
coinbase	当前的挖矿收益
timestamp	自创始以来、以秒计算的当前区块的时间戳
number	当前区块的编号
difficulty	当前区块的难度值
gaslimit	当前区块的 Gas 限值

8.4.4　访问外部变量和函数

Solidity 变量和其他标识符可以简单地通过它们的名称进行访问，但对于内存型变量而言，这种方式会把变量的地址而不是变量值压入堆栈中。存储型变量则不同，在存储器上的变量值很可能没有占满全部的存储块，所以它的“地址”由一个存储块和该存储块内的 1 字节偏移值组成，要读取这个由 x 变量指

仅供非商业用途或交流学习使用

向的存储块，需使用“x_slot”；要读取偏移量，则使用“x_offset”。

内联汇编也可以访问外部函数，在访问时，汇编会向堆栈中压入这些外部函数的“进入标签”，在 Solidity 中的语法如下。

（1）调用函数压入标签。

```
return label,arg1,arg2,…,argn
```

（2）调用返回参数的形式如下：

```
ret1,ret2,…,retm
```

由于调用期间堆栈的偏移经常改变，所以对局部变量的引用很容易出错，如下面的例子：

```
pragma solidity ^0.4.11;

contract C {
    uint b;
    function f(uint x) public returns (uint r) {
        assembly {
            r := mul(x, sload(b_slot))
        }
    }
}
```

8.4.5 本地汇编变量的声明

我们可以使用 let 关键字来声明一个变量，它只在内联汇编中可见，在当前以花括号括起的代码块中有效。let 指令将自动创建一个堆栈，专用于该变量，并且在合约执行到末尾时自动移除。对于这样的变量，必须给定一个初始值，可以全部为 0，也可以是复杂的函数类型表达式，例如下面的程序：

```
pragma solidity ^0.4.16;

contract C {
    function f(uint x) public view returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
```

仅供非商业用途或交流学习使用

```
            b := y
        } // y 的空间在此处释放
        b := add(b, v)
    } // v 的空间在此处释放
  }
}
```

8.4.6　赋值

不论是对本地汇编型变量进行赋值，还是对本地函数型变量进行赋值，都是符合规范的，但在赋值时要留心是内存变量还是存储变量，只能改变指针而不能改变数值。

赋值分为两类：函数型赋值和指令型赋值。函数型赋值的形式类似这样："variable := value"，赋值时需要提供函数型表达式中的一个值，这个值可以产生一个实际的堆栈值。对于指令型赋值，如"=: variable"，变量是直接从堆栈顶端取值的。在这两种方式下，变量名前后都要有冒号。赋值的执行过程就是用新的变量值取代堆栈中的原有变量值。

例如：

```
{
    let v := 0  // 函数型赋值，同时也是变量声明的组成部分
    let g := add(v, 2)
    sload(10)
    =: v        // 指令型赋值，将 sload(10) 的结果赋给 v
}
```

需要注意的是，指令型赋值目前已经过时了。

8.4.7　if 语句

if 语句用于条件执行代码，但是与其他语言不同，if 语句后面没有 else 部分。如果开发者需要多重选项，则需要考虑使用 switch 语句。if 语句的形式如下：

```
{
    if eq(value, 0) { revert(0, 0) }
}
```

需要注意的是，if 语句后面的花括号内容是必不可少的。

仅供非商业用途或交流学习使用

8.4.8 switch 语句

switch 语句可用作基本的“if/else”语句，它从一个表达式中取值，并与几个常量进行比较，根据所符合的常量执行相应的动作。与其他编程语言中易出错的操作不同的是，switch 的控制流并不是从一个选项继续走到下一个选项，而是可以用 default 的形式执行回滚或默认选项的操作。例如：

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

在代码中，选项列表不需要花括号，但是选项体（如 case 0 或 default）必须有花括号。

8.4.9 循环

Solidity 汇编语言支持简单的 for 循环语句。for 循环语句由 3 部分组成：包含初始值的头部、条件部分，以及后面重复执行的循环内容。条件部分必须是函数类型的表达式，而其他两部分必须是代码块。在初始部分声明的任何变量，其范围都扩展到整个循环体（包括条件、重复执行代码）。

下面的例子是对一个内存区域进行求和计算。

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

for 循环也可以写成类似于 while 循环的形式，只需将初始部分、重复执行循环部分保持为空即可。例如：

仅供非商业用途或交流学习使用

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {     // 相当于 while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

8.4.10　函数

Solidity 汇编语言允许定义低阶函数，这些函数既可以从堆栈中取参数值（并返回代码的当前位置），也可以把结果压入堆栈中。调用函数的方式与执行函数型操作码的方式相同。

函数可以在任意位置定义，在包含函数声明的代码段的全部范围内，函数是可见的。在函数内部，不能访问该函数在外部定义的局部变量。

函数也没有明确的 return 语句。如果在调用函数时需要返回多元值，则必须对它们用元组进行赋值，如 a, b := f(x)或者 let a, b := f(x)。

下面的例子执行了乘方的函数。

```
{
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

8.4.11　独立汇编

前文所述的内联汇编语言也可以独立使用。事实上，Solidity 编译器计划将其作为中间语言使用。在这种形式下，汇编语言需要达到如下目标：

仅供非商业用途或交流学习使用

（1）编写的代码应该是可读的，即使代码是由某种 Solidity 编译器生成的。

（2）从汇编语言向二进制代码的转换应该尽可能地少出意外。

（3）控制流容易探测，以便正式校验和优化。

为了实现上述第一个和第三个目标，Solidity 汇编语言提供了高阶构造，如 for 循环、if 语句、switch 语句，以及函数调用。因此，在使用汇编语言编程时可以确定不使用 SWAP、DUP、JUMP 和 JUMPI 这 4 种语句，因为这 4 种语句的前两种会产生数据流混乱，后两种会产生控制流混乱。另外，相比纯操作码语句（如“7 y x add mul”），在汇编语言中优先使用函数型语句（如“mul(add(x, y), 7)”），因为后者更容易清楚地看到操作码与操作数的对应情况。

要实现第二个目标，采用的方式是：更频繁地使用高阶的二进制代码构造器进行编译。由汇编器执行的唯一的非本地操作是对用户定义标识符（如函数、变量等）进行名称查找，这一操作伴随着很简单和常用的作用域规则，同时将从堆栈中清空局部变量。

作用域规则：使用标签、变量、函数及集合进行声明的标识符只在声明它的代码段中可见，当然也包括当前代码段的嵌套代码段。跨越函数边界访问局部变量是不合法的，即使这些局部变量在作用域内也不行，同时不允许使用投影。局部变量在声明之前是不可访问的，但可以访问标签、函数和集合。集合是一种特殊的代码段，用于返回运行时间代码或创建合约，子集合对源自外部集合的任何标识符都不可见。

如果控制流到达代码段的结尾，pop 指令就会插入，以匹配代码段中声明的局部变量的数量。无论何时引用局部变量，代码生成器都需要知道它在堆栈中的当前相对位置，因此需要追踪当前的堆栈高度。由于在代码段的结尾处所有的局部变量都要被移除，因此在执行这一代码段之前和之后的堆栈高度是相同的。如果实际情况不是这样的，系统就会发出警告。

TIPS:

应用 switch 语句、for 语句和函数就可以在不使用手动 JUMP 或 JUMPI 的情况下编写复杂代码，其好处是可以更轻松地分析控制流，为后期的校验和优化带来好处。

仅供非商业用途或交流学习使用

另外，如果允许手动跳转，那么计算堆栈高度就变得相当复杂，还需要知道所有局部变量在堆栈中的位置，否则既不能引用局部变量，也不能在程序段到达结尾时自动移除所有的局部变量。

通过下面的一段 Solidity 程序，我们来了解汇编过程。

原 Solidity 程序代码如下：

```
pragma solidity ^0.4.16;

contract C {
  function f(uint x) public pure returns (uint y) {
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
  }
}
```

汇编后的 Solidity 程序代码如下：

```
{
  mstore(0x40, 0x60) // 存储空闲内存指针
  // 函数分配器
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b {
    let r := f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
    return(ret, 0x20)
  }
  default { revert(0, 0) }
  // 内存配置
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // 合约函数
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

chapter NINE

第 9 章

采用 Solidity 语言开发以太坊游戏

第 8 章介绍了当前极为流行的以太坊编程开发语言——Solidity 的基础知识和组成元素。打一个比方，我们现在已经基本掌握了采用 Solidity 语言来建造一座大厦的各种基础材料和组件，如钢筋、玻璃、砖、混凝土、装饰材料等，而我们的目的是应用这些基础材料建造出符合自己需要的漂亮的房屋。当然这个过程也不是一蹴而就的，我们要学习和借鉴其他人怎样用这些基础材料建造出漂亮的房屋，从而进一步掌握 Solidity 语言各种基本要素的灵活应用。

本章我们将以一个采用 Solidity 语言编制的以太坊游戏作为案例，探索第 8 章介绍的各种基础元素的灵活应用，从而掌握 Solidity 语言，逐步开发出自己的以太坊应用。

9.1 以太坊游戏的特点

以太坊游戏首先是基于互联网的网络游戏，从这一点来说，它与其他网络游戏是有共性的，比如都需要通过互联网执行游戏的进程（与单机游戏不同），借助网络，游戏玩家可以充分交流、创建社区；其次，它也是游戏的一种，游戏运营的通用特点，比如玩家在游戏中产生的虚拟资产（道具、游戏币等），以及相应的激励机制等也都存在。

但同时，以太坊游戏的生态基础是区块链技术，区块链的最大特点是去中心化、交易信息无法被篡改；在区块链的基础上，以太坊游戏还依托以太坊智能合约，而传统的网络游戏或手游是建立在中心化的游戏服务器基础上的，所以对于以太坊游戏来说，它的特点与一般的网络游戏、手游等也存在很大的不同。

目前以太坊技术应用到游戏中，主要采用以太坊作为公有链，它可以很好地解决当前网络游戏生态中的一些困扰和障碍，其中最突出的是两方面：游戏发行，以及游戏资产流通和交易。

1. 游戏发行

当前由于互联网是一个中心化的网络，所以和其他大部分互联网商业形态一样，游戏行业也需要通过大量的中间环节才能到达玩家面前。一款新游戏被开发出来后，首先要由游戏发行商或发行平台向全世界进行发行或推广，吸引更多的玩家来参与游戏。一般情况下，这个中间环节要从游戏总收入中切走 30%的蛋糕。

此外，现在的游戏平台上充斥着各种各样的网络游戏，为了在大量的游戏提供者中吸引用户的眼球，促进下载和应用，游戏开发者需要到处投放广告，这一部分投入占了游戏收入的较大部分。

在发行环节存在着这样的大额资金消耗，对于很多中小型游戏开发者来说显得力不从心，因此，虽然网络游戏行业十分发达，但最终能够占领市场领先地位、被大多数用户看到的游戏只可能是由大资本支持或投资开发出来的游戏，中小游戏开发者的机会渺茫，最后就形成了大公司对网络游戏行业的事实垄断。至于开发什么游戏、发行什么游戏，显然是由资本、中心化的发行平台及大游戏公司来控制的，这是一种典型的中心化结构。

当区块链技术被引入游戏行业后，以太坊及智能合约与游戏行业的结合会彻底改变这种模式，不再需要中心化的游戏发行平台，每款游戏从上线开始就直接面向全部玩家，玩家的体验和口碑决定了游戏的发展状况，而不再是由广告、也不再是由发行公司说了算的。

2. 游戏资产流通和交易

每款网络游戏都包含积分、经验值等基本元素，也包括玩家在玩游戏过程

中收获的游戏币、各种游戏装备等虚拟资产，这些东西在真实世界中也许一文不值，但在游戏世界里，在游戏玩家之间却有着不同寻常的价值。以前的网络游戏由于全部采用中心化的服务器/客户端模式，所以每个玩家的档案都被存放在游戏的服务器上，一方面，需要网络游戏公司投入重金建立大规模的服务器，保证玩家的游戏体验；另一方面，一旦遭遇黑客攻击，或者被游戏公司恶意删档，则玩家存放在服务器上的数据将会消失或被篡改，却无法申辩。

在采用以太坊开发的区块链游戏中，游戏玩家的所有虚拟资产，包括游戏币、游戏装备、经验值等都作为数字资产基于以太坊网络来进行流通和交易，流通更便捷、成本更低，开发者还可以发布游戏专用的 Token（令牌），或者支持现有的其他主流虚拟货币，所有的游戏数据、交易信息等都被写入以太坊，利用区块链无法被篡改的特点，实现玩家对游戏资产的全面掌控。

9.2 以太坊游戏开发准备

以太坊游戏是在以太坊网络基础上开发出来的以太坊应用程序，当然首先具备以太坊的特点，即以智能合约为主要载体。

其次，需要用到针对以太坊的应用开发平台，比如 Geth 客户端、Node.js、Gonache，以及浏览器集成开发环境 Remix 等；可以使用的语言也包括 JavaScript、Golang，以及我们主要介绍的 Solidity 语言。事实上，由于 JavaScript 语言在区块链出现之前有着广泛的应用，所以自转换到区块链开发，尤其是以太坊开发以来，采用 JavaScript 语言开发的以太坊应用是数量最多的；但是也可以看到，随着 Solidity 语言的快速发展，越来越多的开发者转移到 Solidity 平台上来。

要深入学习和借鉴其他开发者应用 Solidity 语言开发 DApp 的经验，我们需要借助一个开源社区的力量，这就是 GitHub。

GitHub 是全球软件开发者汇集起来的一个领先开发平台，创建于 2007 年，现在全球范围内有 2800 万人利用这个社区进行软件开发，包括学习、交流及共同工作，在 GitHub 上已经有 8500 万个应用开发资源库，也有大量的开源软件。

TIPS:

> 随着区块链技术的日益发展，这 8500 万个应用开发资源库中也出现了大量的以太坊开发 DApp 资源库，可以作为我们学习和掌握 Solidity 语言的鲜活资料。

所以，进行以太坊游戏开发，我们首先要注册 GitHub 账号，进入这个开源社区，一方面学习优秀开发者的经验，另一方面可以把自己的以太坊 DApp 存储到 GitHub 上，形成自己的资源库，便于自己的团队日后协同工作。

其次，我们要准备好自己的开发平台。在此推荐采用 Remix IDE，因为它无须安装即可使用，对于初学者来说界面更加友好，对于具有较高开发水平的程序员来说依然可以满足开发要求。当然，其他平台也非常好，有可能效率更高，比如 Geth 的命令行模式，不少传统的程序员更喜欢命令行模式，因为能够感受到对于代码的掌控力。还有一些开发者喜欢在 Linux 系统下开发，这需要用到 Ubuntu 平台。其他一些开发平台，如 Gonache 等，甚至 Mist 钱包软件的代码测试界面，也都具有各自的特点，开发者可以依据自己的感受确定自己最偏爱的开发平台。本章讲解所用的平台为 Remix。

如果使用其他的开发平台，则需要按第 7 章中所介绍的内容，在本机上安装相关的平台程序及相应的安装包，并进行测试，确保各平台正常运行。

其他方面的准备还包括对区块链基础知识的储备，以及编程的基础知识。有了这些基础知识的支撑，我们进入以太坊的编程中，会更加容易取得较好的效果。

TIPS:

> 由于 Solidity 语言与 JavaScript 语言、Python 语言有较深厚的渊源，所以，如果开发者原来有较好的 JavaScript 或 Python 语言基础，则能够较好地促进用 Solidity 语言开发以太坊 DApp 的进程。

另外，还有一项非常重要的准备：以太币。开发者在开发以太坊游戏的过程中，在编程、测试阶段可以采用测试网络，各种开发平台都提供了相应的测试网络连接选项；但是当测试完成后，如果需要部署到以太坊主网络上，那么这一环节需要消耗以太币作为 Gas 费用。如果没有以太币，那么部署合约的行

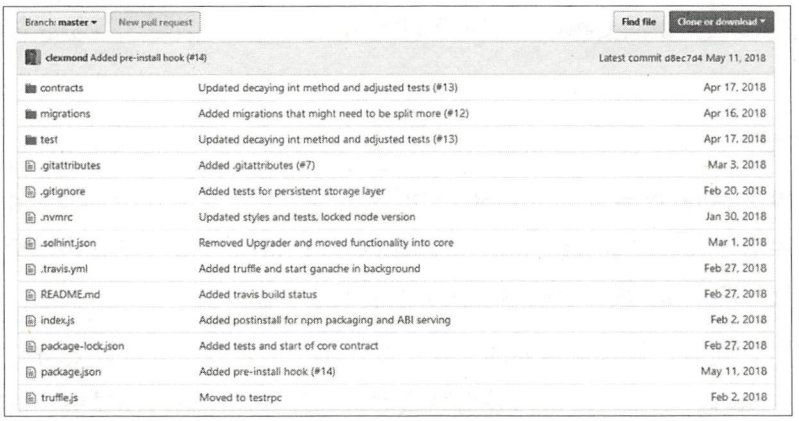
为不会得到以太坊矿工的确认，部署难以实现。因此，开发者需要有自己的以太币钱包（包括以太币账户、公钥、私钥等），并通过挖矿或购买、转账等方式取得以太币，使开发成功的以太坊 DApp 能够成功部署到以太坊主网络上。

9.3 以太坊游戏 Influence 代码框架

在这一节中，我们以来自 GitHub 的一个以太坊游戏 DApp——Influence（影响力）作为案例，进入以太坊开发区块链游戏的内核进行探索。

Influence 是一款大型的战略游戏，由 Chris Lexmond 开发，他是 RISE Cooperative 公司的创始人，也是 Influence 游戏的开发者，还是一家商业地产公司 REscour 的联合创始人。在 GitHub 上，其资源库名称为 clexmond，网址为 https://github.com/clexmond/influence-dapp。根据作者提供的游戏简介，这是一款建立在小行星带上的大型战略模拟游戏，游戏网页为 https://www. influenceth.io/。不过从网站上来看，该游戏还处于开发阶段，并没有进入实际运营。这一点并不影响我们学习一个高水平开发者对于 Solidity 语言的应用过程。

进入 GitHub 后，我们可以看到整个资源库目前拥有如图 9.1 所示的代码文件。



Branch: master ▾ | New pull request | Find file | Clone or download ▾

clexmond Added pre-install hook (#14) — Latest commit d8ec7d4 May 11, 2018

contracts	Updated decaying int method and adjusted tests (#13)	Apr 17, 2018
migrations	Added migrations that might need to be split more (#12)	Apr 16, 2018
test	Updated decaying int method and adjusted tests (#13)	Apr 17, 2018
.gitattributes	Added .gitattributes (#7)	Mar 3, 2018
.gitignore	Added tests for persistent storage layer	Feb 20, 2018
.nvmrc	Updated styles and tests, locked node version	Jan 30, 2018
.solhint.json	Removed Upgrader and moved functionality into core	Mar 1, 2018
.travis.yml	Added truffle and start ganache in background	Feb 27, 2018
README.md	Added travis build status	Feb 27, 2018
index.js	Added postinstall for npm packaging and ABI serving	Feb 2, 2018
package-lock.json	Added tests and start of core contract	Feb 27, 2018
package.json	Added pre-install hook (#14)	May 11, 2018
truffle.js	Moved to testrpc	Feb 2, 2018

图 9.1 Influence 游戏的代码文件

从图 9.1 中可以看出，现有的代码文件分为如下几个不同的组成部分。

（1）智能合约 contracts。这是本款游戏的主要组成部分，也是核心代码，用来实现游戏的各种主要功能，比如界面、游戏操作功能、游戏库函数、游戏内部的各种模块，对于这一款以太坊游戏来说，还有拍卖、发行游戏 Token 等功能。

进入 contracts 子文件夹后，可以看到目前已经开发出来的各类智能合约，如图 9.2 所示。

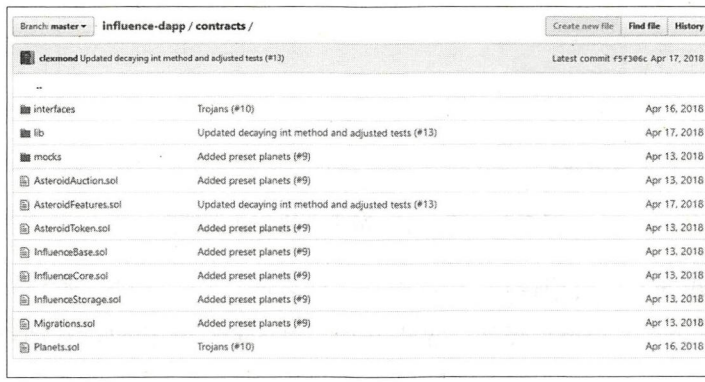


图 9.2　Influence 游戏中的智能合约

（2）migrations（迁移类）。这一部分内容借鉴的是其他语言提供的功能模块。为了提高开发效率，对于一些相同功能的模块，可以借鉴其他语言中的公用功能模块，而不用自己去重新开发公用内容，也就是说不需要“重新发明轮子”，直接借鉴即可，这些模块统一放在迁移类 migrations 中。如图 9.3 所示是 Influence 游戏中的迁移类文件夹详情，可以看到其中有 7 个.js 文件，这些都是 JavaScript 脚本文件，因为 Solidity 语言与 JavaScript 语言有很强的兼容性，所以可以直接进行迁移使用。

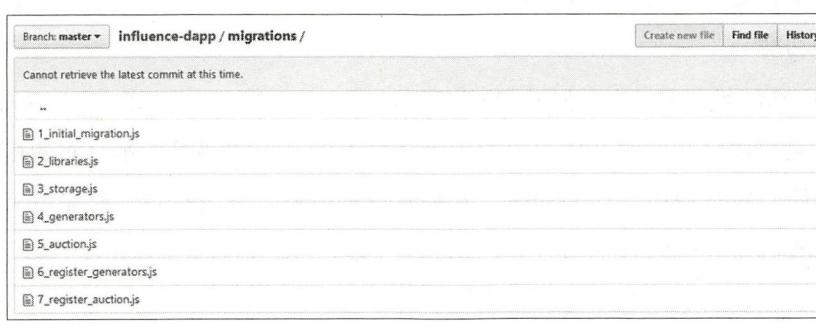


图 9.3　Influence 游戏中的迁移类文件夹详情

（3）测试文档 test 文件夹。其中存储的是帮助文件、库文件，以及对核心功能进行测试的文件，如图 9.4 所示。

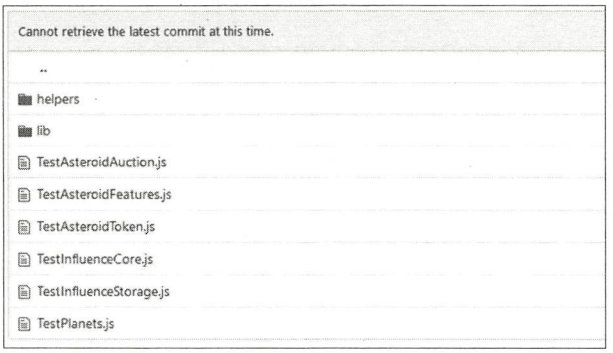


图 9.4　Influence 游戏中的 test 文件夹

（4）游戏开发的公用模块。其特点是文件名前以“.”为引导，如“.gitattributes”。在资源库中共有 5 个公用模块（见表 9.1），供 contracts 代码进行调用。

表 9.1　资源库中的 5 个公用模块

模块名称	内　容	备　注
.gitattributes	在 git 中设置 Solidity 语言属性	
.gitignore	在 git 中忽略特殊文件	
.nvmrc	Node.js 的版本管理器	长效支持 lts
.solhint.json	一种静态检查器，提供安全和风格指南，以及智能合约验证的最佳实践规则	
.travis.yml	对开源项目进行 Travis 持续集成	供 GitHub 开源项目使用

从表 9.1 中可以看出，这些公用模块文件的功能集中于 Solidity 语言的 DApp 与其他平台之间的切换和交流，以及对开发项目本身的管理、开发风格、系统集成等功能。

（5）功能性代码文件，如 readme.mp、index.js、truffle.js 等。其中，readme.mp 是每个应用程序必备的文档，它的内容详略不同，但主要目的都是对应用程序进行说明，包括应用程序的功能、安装及使用流程、开发特点、联系方式，以及开发者想要介绍的其他内容等；index.js 的内容是对智能合约进行 ABI（应用程序二进制接口）的索引，相当于应用程序与底层代码的接口；truffle.js 则是对 Truffle 开发平台进行调用的配置文件。

（6）支持包，即 package 文件。由于 Solidity 语言源自 JavaScript 语言，因此有不少的地方还需要下载或使用原来的软件包（package）。而且在编程与开发过程中对软件包的使用也会发生变化，因此需要两个文件：package.json 和 package-lock.json，其中后者是 package.json 的校验版本。两个文件中对于同样的软件包的版本号总是保持一致的，如果开发中更新了 package.json 中某个软件包的版本，那么在执行 npm 命令时会下载最新的软件包并更新 package-lock.json 文件中的版本号；如果两个文件中的版本号一致，就不需要下载最新的软件包。

9.4　以太坊游戏 Influence 源代码解读

在上一节中我们介绍了以太坊游戏的特点，以及目前开发中的 Influence 游戏的代码结构，本节将进入 Influence 游戏的核心功能代码，通过对源代码的分析和解读，逐步了解和掌握使用 Solidity 语言开发以太坊游戏的能力。

在进入 Influence 的 GitHub 资源库后，我们将重点放在核心代码——contracts 文件夹上。在进入该文件夹后，可以看到其核心代码有如图 9.5 所示的文件。

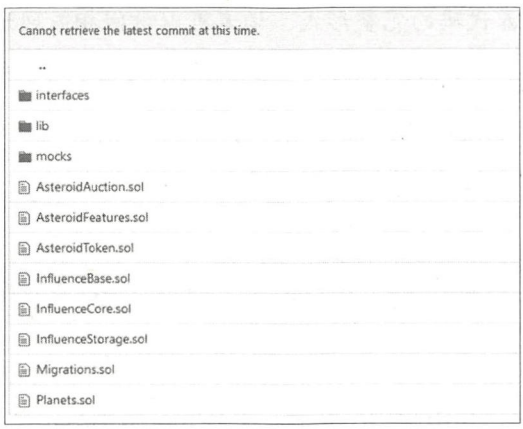


图 9.5　Influence 资源库的合约内容

从 contracts 文件夹中可以看到，除 8 个 Solidity 文件外，还有 3 个次级文件夹。这 8 个文件分别实现游戏某一方面的核心功能，如拍卖、代币功能；3

个次级文件夹对应的分别是游戏界面、库函数、模型文件夹，说明其中分类存放了对应功能的程序代码。

9.4.1 游戏界面

进入 interfaces（界面）文件夹，如图 9.6 所示。

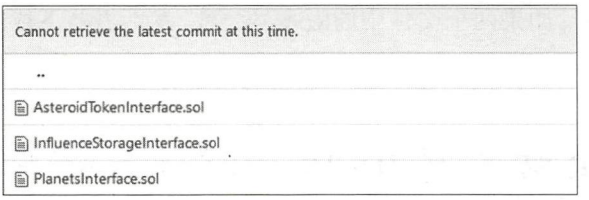


图 9.6 interfaces 文件夹

其中包括 3 个 Solidity 文件：AsteroidTokenInterface.sol，这是游戏 Token 的界面文件；InfluenceStorageInterface.sol，这是游戏数据存储的界面文件；PlanetsInterface.sol，这是游戏中各种行星的界面文件。

我们打开 AsteroidTokenInterface.sol 文件，可以看到游戏 Token 界面的程序代码（作者注：为方便读者阅读和理解源代码，我们将直接在代码段后面进行解读。这可能会影响代码的形式完整，但更有助于对代码的深入了解。如需掌握源代码的完整形式，读者可以访问相关网址）。

```
pragma solidity ^0.4.21;
```

【解读】每个 Solidity 程序都要先进行 Solidity 语言的版本声明。源代码为“pragma solidity ^0.4.21;”，说明采用了 Solidity 0.4.21 版本进行开发。

```
/**
 * @title AsteroidToken interface that matches ERC721
 * @dev see https://github.com/ethereum/eips/issues/721. Provided by
OpenZeppelin.
 */
```

【解读】从注释语句中我们了解到，这个游戏的 Token 是采用 ERC721 规则发行的，其名称为 AsteroidToken（小行星币）。而 ERC721 的详情参考网站是 https://github.com/ethereum/eips/issues/721，它是由 OpenZeppelin 这个开源的智能合约函数库提供的。

```
contract AsteroidTokenInterface {
  event Transfer(address indexed _from, address indexed _to, uint256
_tokenId);
  event Approval(address indexed _owner, address indexed _approved,
uint256 _tokenId);
```

【解读】整段代码创建了一个界面合约 AsteroidTokenInterface，在合约中定义了两个事件（event）、在 ERC721 规则下的 5 个函数，以及在非 ERC721 规则下的两个函数。

两个事件分别为 Transfer（转账）和 Approval（认可）。在 Transfer 事件中，包括 3 个参数：①转出账户地址，即 address indexed _from，变量类型为 address；②转入账户地址，即 address indexed _to，变量类型为 address；③转账的 Token 的数量，是一个无符号整数，即 uint256 _tokenId。

在 Approval 事件中，也包括 3 个参数：①所有者的账户地址，即 address indexed _owner，是一个地址类型的变量；②被认可的转出地址，即 address indexed _approved；③获得认可的 Token 的数量，是一个无符号整数，即 uint256 _tokenId。

```
  function balanceOf(address _owner) public view returns (uint256
_balance);
  function ownerOf(uint256 _tokenId) public view returns (address
_owner);
  function transfer(address _to, uint256 _tokenId) public;
  function approve(address _to, uint256 _tokenId) public;
  function takeOwnership(uint256 _tokenId) public;
```

【解读】在合约中声明了在 ERC721 规则下的 5 个函数。

（1）账户余额函数 balanceOf()：返回一个 uint256 类型的账户余额，函数参数为所有者的账户地址（address_owner）。这个函数是公共类型（public）的，可以从外部访问；同时它被声明为 view 型，表示其状态是不可修改的。

（2）所有者账户函数 ownerOf()：函数参数为 uint256 类型的 Token 数量，即 uint256 _tokenId，返回所有者账户地址（address _owner）。这个函数也是公共类型（public）的，状态不可修改（view）。

（3）转账函数 transfer()：这个函数是公共类型（public）的，参数有两个，即转入地址（address _to）和转入数量（uint256 _tokenId）。

（4）认可函数 approve()：这个函数是公共类型（public）的，参数有两个，即转入地址（address _to）和转入数量（uint256 _tokenId）。

（5）取得所有权函数 takeOwnership()：这个函数是公共类型（public）的，参数只有一个，即无符号整数 uint256 _tokenId，其功能是对需要转入的 Token 取得所有权。

```
  // Non-ERC721 methods
  function isTokenMinted(uint256 _tokenId) public view returns (bool);
  function mint(address _to, uint256 _tokenId) public;
}
```

【解读】对于在非 ERC721 规则下的 Token，合约中声明了两个函数。

（1）判断 Token 是否已经铸币的函数 isTokenMinted()：该函数为公共类型（public）的，状态不可修改（view），参数为无符号整数 uint256 _tokenId，返回值为布尔型（bool），即只有 1 和 0 两种值。

（2）铸币函数 mint()：该函数为公共类型（public）的，参数有两个，即铸币形成的 Token 所要给予的账户地址（address _to），以及铸币产生的 Token 数量（uint256 _tokenId）。

以上是对 AsteroidTokenInterface.sol 代码文件的解读，另外两个代码文件的内容都是界面合约、合约中函数的声明，不再赘述了。

9.4.2 库文件 lib

我们进入 contracts 下一级的 lib 文件夹，其中有 5 个 Solidity 源代码文件，如图 9.7 所示。

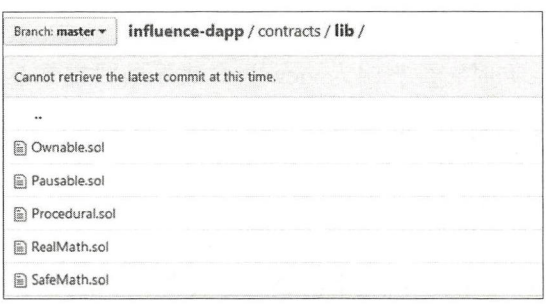


图 9.7 lib 文件夹

仅供非商业用途或交流学习使用

1. 拥有机制 Ownable.sol

对源代码的解读如下。

```
pragma solidity ^0.4.21;
```

【解读】Solidity 语言版本号声明。

```
/**
 * @dev The Ownable contract has an owner address, and provides basic
authorization control
 * functions, this simplifies the implementation of "user permissions".
From OpenZeppelin.
 */
```

【解读】注释部分说明了文件的功能。在本文件中定义了 Ownable（可持有）合约，拥有一个所有者账户地址，提供了基本的权限控制功能，简化了“用户许可”功能的执行过程。

```
contract Ownable {
  address public owner;

  event OwnershipTransferred(address indexed previousOwner, address
indexed newOwner);
```

【解读】本段程序代码定义了合约 Ownable，其中首先声明了一个参数 owner，这个参数是地址（address）类型、公共（public）类型的；其次声明了一个事件 OwnershipTransferred，即所有者转移事件，从前一个所有者账户（address indexed previousOwner）转移到新的所有者账户（address indexed newOwner）。对于这两个账户采用了 indexed 属性，这是 event（事件）参数独有的属性，表示该事件的参数在存储时采用的是特殊的 topics 数据结构，可被检索到。

```
  // The Ownable constructor sets the original `owner` of the contract
to the sender account.
  function Ownable() public {
    owner = msg.sender;
  }
```

【解读】本段程序代码定义了无参函数 Ownable()，为公共类型，同时 Ownable 构造器将合约的初始变量 owner 定向到 sender（发送器）账户。

```
  // Throws if called by any account other than the owner.
  modifier onlyOwner() {
```

仅供非商业用途或交流学习使用


```
    require(msg.sender == owner);
    _;
  }
```

【解读】这是一个通用的模块，在很多 Solidity 程序中都可以看到。它用函数修饰符定义了 onlyOwner()无参函数，其中使用 require()请求函数进行格式检查，其含义是：如果 owner 调用这个函数，且与 msg.sender 的内容相符，则函数得以执行，函数本体从“_;”符号处插入执行；否则就会触发异常。

```
  // Allows the current owner to transfer control of the contract to a
newOwner.
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
```

【解读】这一代码段的功能是在现有所有者与新所有者之间进行合约控制权的转让，其中定义了一个控制权转换函数 transferOwnership()，其参数是新所有者的账户地址（adderss newOwner），这是一个公共类型的函数，并调用前面定义的 onlyOwner()函数。

控制权转换函数的内容是：先用 require()函数进行格式检查，确认 newOwner 的地址不能为 0，当条件满足后释放控制权转让事件，其参数包括原所有者、新所有者的值，同时将新所有者的值赋给 owner 变量，实现了新所有者对合约控制权的占有。

2. 暂停机制 Pausable.sol

对源代码的解读如下。

```
pragma solidity ^0.4.21;

import "./Ownable.sol";
```

【解读】此处除常见的版本号外，还有一个导入文件的命令 import "./Ownable.sol"，它的作用是将当前文件夹下的 Ownable.sol 文件内容导入本文件中来。

仅供非商业用途或交流学习使用


```
/**
 * @dev Base contract which allows children to implement an emergency
stop mechanism. Provided by OpenZeppelin.
 */
```

【解读】这一段分行注释说明了暂停机制的目的：当儿童玩这款游戏时，可以实现紧急暂停。

```
contract Pausable is Ownable {
  event Paused();
  event Unpaused();

  bool public paused = false;
```

【解读】本段程序代码定义了一个合约 Pausable，它是从合约 Ownable 中派生出来的，其中声明了两个事件 Paused()和 Unpaused()，另外声明了一个布尔型的公共变量 paused，初始值赋为 false。

```
  /**
   * @dev Modifier to make a function callable only when the contract
is not paused.
   */
  modifier whenNotPaused() {
    require(!paused);
    _;
  }
```

【解读】本段程序代码的功能是当合约处于未暂停状态时，使用修饰符调用函数 whenNotPaused()。在函数调用过程中，先由 require()函数自动检查函数的前置条件是否为 true，如果满足，则将函数本体从“_;”处插入。

```
  /**
   * @dev Modifier to make a function callable only when the contract
is paused.
   */
  modifier whenPaused() {
    require(paused);
    _;
  }
```

【解读】当合约处于暂停状态时，使用修饰符调用函数 whenPaused()。在函数调用过程中，先由 require()函数自动检查函数的前置条件是否为 true，如果满足，则将函数本体从“_;”处插入。

仅供非商业用途或交流学习使用


```
  /**
   * @dev called by the owner to pause, triggers stopped state
   */
  function pause() public onlyOwner whenNotPaused {
    paused = true;
    emit Paused();
  }
```

【解读】本段程序代码定义了公共类型的函数 pause()，它是从函数 onlyOwner()和函数 whenNotPaused()中继承而来的，除拥有父函数的变量和功能外，还增加了对 paused 变量的赋值，将 paused 赋为 true；然后释放 Paused()事件，由所有者 owner 调用 pause 状态，触发停顿状态。

```
  /**
   * @dev called by the owner to unpause, returns to normal state
   */
  function unpause() public onlyOwner whenPaused {
    paused = false;
    emit Unpaused();
  }
}
```

【解读】本段程序代码定义了公共类型的函数 unpause()，它是从函数 onlyOwner()和函数 whenPaused()中继承而来的，除拥有父函数的变量和功能外，还将 paused 变量赋值为 false；然后释放 Unpaused()事件，由所有者 owner 调用未暂停状态，返回合约的正常运行状态。

3. 游戏进程文件 Procedural.sol

对源代码的解读如下。

```
pragma solidity ^0.4.21;

import "./RealMath.sol";
```

【解读】除 Solidity 语言版本号声明之外，还导入了同一文件夹下的 RealMath.sol 文件，这个文件的主要内容是数学计算功能函数。

```
library Procedural {
  using RealMath for *;
```

【解读】本段程序代码定义了一个库函数 Procedural，其内容是将 RealMath

仅供非商业用途或交流学习使用

库中的函数附加给任意类型，也就是说本文件中的任何类型都可以调用 RealMath.sd 文件中的数学计算功能函数。

```
  /**
   * We are going to define a RandNode struct to allow for hash chaining.
   * You can extend a RandNode with a bunch of different stuff and get
a new RandNode.
   * You can then use a RandNode to get a single, repeatable random value.
   * This eliminates the need for concatenating string selfs, which is
a huge pain in Solidity.
   */
  struct RandNode {
    // We hash this together with whatever we're mixing in to get the
child hash.
    bytes32 _hash;
  }
```

【解读】本段程序代码定义了一个结构 RandNode，用来对链式结构进行哈希，既可以用来对单一、可重复的随机值取值，也可以扩展到其他不同的数据，形成一个新的 RandNode 结构。这一点克服了 Solidity 语言在链式数据上的不足。

RandNode 结构的内容是一个 bytes32 字节数组，名为_hash。

```
  /**
   * All the functions that touch RandNodes need to be internal.
   * If you want to pass them in and out of contracts just use the bytes32.
   *
   * You can get all these functions as methods on RandNodes by "using
Procedural for *" in your library/contract.
   *
   * Mix string data into a RandNode. Returns a new RandNode.
   */
  function derive(RandNode _self, string _entropy) internal pure returns
(RandNode) {
    // Hash what's there now with the new stuff.
    return RandNode(sha256(_self._hash, _entropy));
  }
```

【解读】本段程序代码定义了一个内部函数 derive()，它的参数是一个 RandNode 结构，名为_self，另一个是字符串_entropy，它的返回值是一个自变量(用 pure 来增加的属性，表示既不能读取也不能修改状态)RandNode。derive()函数的具体实施代码就是先调用 sha256()函数将_self 中的哈希值与字符串

仅供非商业用途或交流学习使用

_entropy 合并起来进行再次哈希，然后传递给 RandNode，并返回这个新产生的 RandNode 结构。

```
  // Mix signed int data into a RandNode. Returns a new RandNode.
  function derive(RandNode _self, int256 _entropy) internal pure returns
(RandNode) {
    return RandNode(sha256(_self._hash, _entropy));
  }

  // Mix unsigned int data into a RandNode. Returns a new RandNode.
  function derive(RandNode _self, uint256 _entropy) internal pure
returns (RandNode) {
    return RandNode(sha256(_self._hash, _entropy));
  }
```

【解读】本段程序代码分别定义了不同参数的 derive()函数，功能相似，分别是对 int256 和 uint256 两个类型的变量与原来的 RandNode 进行混合哈希，求得新的 RandNode 结构。

```
  /**
   * Returns the base pseudorandom hash for the given RandNode.
   * Does another round of hashing in case you made a RandNode("Stuff").
   */
  function getHash(RandNode _self) internal pure returns (bytes32) {
    return sha256(_self._hash);
  }
```

【解读】本段程序代码定义了内部函数 getHash()，其参数是 RandNode 数据结构_self，返回值是一个自变量：32 位的字节数组。函数内的执行语句是调用 sha256()函数对_self 结构中的原哈希值进行再次哈希，再造了一个 RandNode 结构。

```
  // Return true or false with 50% probability.
  function getBool(RandNode _self) internal pure returns (bool) {
    return getHash(_self) & 0x1 > 0;
  }

  // Get an int128 full of random bits.
  function getInt128(RandNode _self) internal pure returns (int128) {
    // Just cast to int and truncate
    return int128(int256(getHash(_self)));
  }
```

仅供非商业用途或交流学习使用


```
  // Get a real88x40 between 0 (inclusive) and 1 (exclusive).
  function getReal(RandNode _self) internal pure returns (int128) {
    return getInt128(_self).fpart();
  }

  // Get an integer between low, inclusive, and high, exclusive.
Represented as a normal int, not a real.
  function getIntBetween(RandNode _self, int88 _low, int88 _high)
internal pure returns (int88) {
    int128 range = RealMath.toReal(_high) - RealMath.toReal(_low);
    int128 result = RealMath.mul(getReal(_self), range) +
RealMath.toReal(_low);
    return RealMath.fromReal(RealMath.ipart(result));
  }
```

【解读】以上定义的 4 个函数分别用于对 RandNode 结构的_self 变量进行不同类型的数学处理，有的函数调用了 RealMath.sol 文件中的数学计算功能函数，因为 RealMath.sol 已经导入本文件，所以可以正常调用。

```
  // Returns a normal int (roughly) normally distributed value between
low and high
  function getNormalIntBetween(RandNode _self, int88 _low, int88 _high)
internal pure returns (int88) {
    int88 accumulator = 0;

    for (uint i = 0; i < 5; i++) {
      accumulator += getIntBetween(derive(_self, i), _low, _high);
    }

    return
RealMath.fromReal(RealMath.div(RealMath.toReal(accumulator),
RealMath.toReal(5)));
  }
```

【解读】本段程序代码定义了内部函数 getNormalIntBetween()，返回一个特定范围内的呈正态分布的整数值，参数为 RandNode 结构的变量_self，分布范围低限为 int88 _low，分布范围高限为 int88 _high，返回变量是一个 int88 类型的值。

在函数的执行语句中，首先定义了一个 int88 类型的累加器 accumulator，并赋初始值为 0。然后开始 5 次循环，对 accumulator 进行累加，累加值则调用 getIntBetween()函数，并嵌套调用 derive()函数，对_self、分布范围的高限和低

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

限 3 个参数进行哈希，取得最终的 accumulator 值。

在处理完 accumulator 值之后，再调用 RealMath.sol 中的多个数学计算功能函数，以 accumulator 为参数进行嵌套运算，计算出返回值。

```
  /**
   * "Folds" a normal int distribution in half to generate an approx decay
function
   * Only takes a high value (exclusive) as the simplistic approximation
relies on low being zero
   * Returns a normal int, not a real
   */
  function getDecayingIntBelow(RandNode _self, int88 _high) internal
pure returns (int88) {
    int128 normalInt = RealMath.toReal(getNormalIntBetween(_self, 0,
_high * 2 - 1));
    int128 adjusted = normalInt - RealMath.toReal(_high);
    return RealMath.fromReal(adjusted.abs());
  }
```

【解读】本段程序代码定义了一个内部函数 getDecayingIntBelow()，封装一半的整数分布范围，使函数接近于衰变函数，函数的参数只对分布范围的高限进行处理，返回一个自变量，其类型是正常的 int88 型整数值而非实数。

在函数的执行语句中，首先定义了 int128 类型的变量 normalInt，嵌套调用了 RealMath.sol 中的函数和前面程序中定义的 getNormalIntBetween()函数，计算出的实数值转换为 128 位整数型赋给 normalInt。

然后用 normalInt 与分布范围高限值相减，得出一个 int128 位的调整值 adjusted，对其求绝对值后，转换为 int88 类型返回。

```
  /**
   * Get a real between realLow (inclusive) and realHigh (exclusive).
   * Only actually has the bits of entropy from getReal, so some values
will not occur.
   */
  function getRealBetween(RandNode _self, int128 _realLow, int128
_realHigh) internal pure returns (int128) {
    return getReal(_self).mul(_realHigh - _realLow) + _realLow;
  }
```

【解读】本段程序代码定义了内部函数 getRealBetween()，其功能是对分布

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

范围的低限和高限进行实数取值，然后在返回语句中进行数学运算：调用前面定义的 getReal 函数()对高、低限值做相减计算，结果再与低限值相加，并返回最终结果（以 int128 类型返回）。

```
  // Roll a number of die of the given size, add/subtract a bonus, and
return the result. Max size is 100.
  function d(RandNode _self, int8 _count, int8 _size, int8 _bonus)
internal pure returns (int16) {
    if (_count == 1) {
      // Base case
      return int16(getIntBetween(_self, 1, _size)) + _bonus;
    } else {
      // Loop and sum
      int16 sum = _bonus;

      for (int8 i = 0; i < _count; i++) {
        // Roll each die with no bonus
        sum += d(derive(_self, i), 1, _size, 0);
      }

      return sum;
    }
  }
}
```

【解读】本段程序代码定义了一个内部函数 d()，其参数包括：RandNode 结构的变量_self，int8 类型的变量 count（计数器）、size（给定的区域大小）、bonus（游戏奖励），返回值是一个 int16 类型的自变量，即根据游戏给定范围的完成情况给予玩家一定的奖励，其中在游戏角色死亡后不给予奖励。最后返回奖励的汇总值。

以上是对库文件中一部分源代码的解读，其他两个库文件均为数学计算函数的文件，由于篇幅所限，我们不再对其进行详细解读，读者可自行查阅。下面将着重介绍游戏的主要功能文件。

9.4.3　游戏主功能：小行星拍卖

在 contracts 文件夹下保存着游戏主要功能模块的 Solidity 文件，我们先了解 AsteroidAuction.sol（小行星拍卖）功能的程序源代码。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

对源代码的解读如下。

```
pragma solidity ^0.4.21;

import "./InfluenceBase.sol";
import "./interfaces/AsteroidTokenInterface.sol";
```

【解读】Solidity 语言版本号声明，以及导入两个文件：当前文件夹下的 InfluenceBase.sol，以及通过文件路径导入的 interfaces 文件夹下的 AsteroidTokinInterface.sol。

```
/**
 * @dev Defines the initial auction of asteroids releasing them in daily
batches.
 */
contract AsteroidAuction is InfluenceBase {
  uint public startTime; // in seconds since epoch
  uint public duration; // in days
  uint public minimumBid; // in Wei

  event AuctionStarted(uint _asteroidId, uint _startedAt);
  event BidEntered(uint _asteroidId, address _by, uint _amount, uint
_enteredAt);
  event AuctionWon(uint _asteroidId, address _by, uint _lastBidAmount,
uint _wonAt);

  function AsteroidAuction(address _storage) InfluenceBase(_storage)
public {
    namespace = "AsteroidAuction";
    version = 1;
  }
```

【解读】本段程序代码定义了“小行星拍卖合约”，这个合约是从合约 InfluenceBase 派生而来的，并且增加声明了一批变量和事件，定义了函数 AsteroidAuction()，它是从 InfluenceBase()函数继承而来的。

声明的 3 个变量为 uint 的公共类型变量，分别是开始时间（以秒为单位）、持续时间（以日为单位）、最小投注（以 wei 为单位）。

声明的 3 个事件分别为拍卖开始（AuctionStarted）、进入投标（BidEntered）、拍卖胜利（AuctionWon）。

仅供非商业用途或交流学习使用

还定义了函数 AsteroidAuction()，参数为存储地址，执行语句为对存储空间进行命名并设置初始版本号为 1。

```
  /**
   * @dev Sets the standard params for the initial auction
   * @param _timestamp Seconds since unix epoch the auction should start
at
   * @param _duration Number of days
   * @param _minimumBid Amount in wei
   */
  function setAuctionParams(uint _timestamp, uint _duration, uint
_minimumBid) external onlyOwner {
    startTime = _timestamp;
    if (startTime < now) startTime = now;
    duration = _duration;
    minimumBid = _minimumBid;
  }
```

【解读】本段程序代码定义了外部函数 setAuctionParams()，这个函数继承自 onlyOwner()函数，自己的参数有 3 个 uint 值：从 UNIX 时间起点以秒计时的拍卖起始时间（_timestamp）、以天数为单位的拍卖持续时间（_duration）、最小投标金额（_minimumBid）。

在函数的执行语句中，将上述 3 个参数的值赋给 3 个变量：startTime、duration 和 minimumBid，并将 startTime 时间与当前时间保持同步。

```
  /**
   * @dev Submit a bid for an asteroid
   * @param _asteroidId Between 1 and TOTAL_ASTEROID. Same as param passed
to AsteroidFeatures.getAsteroidSeed
   */
  function enterBid(uint _asteroidId) external payable {
    require(msg.value >= minimumBid);
    require(now >= availableForAuctionAt(_asteroidId));

    // Ensure that this token has never been minted, and is therefore
un-owned.
    address tokenContractAddress = influenceStorage.getAddress(keccak256
("Contracts", "name", "AsteroidToken"));
    AsteroidTokenInterface tokenContract = AsteroidTokenInterface
(tokenContractAddress);
    require(!tokenContract.isTokenMinted(_asteroidId));
```

仅供非商业用途或交流学习使用

```
    /**
     * The auctions include a 1 day extension every time there's a new
high bid. To push them to actually
     * finish and not have constant small increments we require that each
new bid doubles the previous.
     */
    bytes32 bidKey = keccak256(namespace, "bid", _asteroidId);
    uint priorBid = influenceStorage.getUInt(bidKey);
    require(msg.value >= priorBid * 2);
    bytes32 bidderKey = keccak256(namespace, "bidder", _asteroidId);
    bytes32 timeKey = keccak256(namespace, "bidTime", _asteroidId);

    // If there's a prior bid we need to handle it
    if (priorBid != 0) {
      // Check that the auction hasn't ended (1 day after last bid)
      uint endTime = influenceStorage.getUInt(timeKey) + 86400;
      require(now < endTime);

      // Get the prior bidder's balance and increment it by their lost bid
      address priorBidder = influenceStorage.getAddress(bidderKey);
      bytes32 escrowKey = keccak256(namespace, "escrow", priorBidder);
      uint escrowBalance = influenceStorage.getUInt(escrowKey);
      influenceStorage.setUInt(escrowKey, escrowBalance + priorBid); //
Do we need SafeMath?
    }

    // Finally, record our new bid and fire event
    influenceStorage.setUInt(bidKey, msg.value);
    influenceStorage.setUInt(timeKey, now);
    influenceStorage.setAddress(bidderKey, msg.sender);
    emit BidEntered(_asteroidId, msg.sender, msg.value, now);
  }
```

【解读】以上这段较长的程序代码的主要功能是对拍卖投标函数 enterBid() 进行详细定义。执行过程如下：

首先确定该函数是一个外部函数，并给出了函数属性 payable。有了这个属性，函数就可以接收和支付以太币。

然后使用 require()函数进行自动的函数参数检验，对 msg.value 和 now 的初始值进行检验，确保now的值不低于在当前可用的小行星币的数量_asteroidId

仅供非商业用途或交流学习使用

（1<_asteroidId<小行星币的总数量 TOTAL_ASTEROID）

接下来对投标用的 Token 的状态进行检验，确保这个 Token 是还未铸币的，因此是一种“无产权”的状态。先调用 influenceStorage.getAddress()取得 Token 的合约地址 tokenContractAddress，并通过合约继承方式调用了该 Token 的合约接口；随后采用 require()函数进行检验，确认是否是未被铸币的 Token。

拍卖的规则是：每次可扩展至一天时间，取最高投标值，每次的新投标值必须是前一次投标值的两倍以上。在执行语句上，先取得上一次投标的值，通过 keccak256 转换为 bytes32 类型后再转换为 uint 值并将其赋给 priorBid，此时调用 require()函数检验当前投标值是否是前一次投标值的两倍以上。检验通过后，通过引用存储空间取得当前投标者的投标数量和投标时间，并通过 keccak256 转换为 bytes32 类型。

当存在前一次投标时，需要进行处理。如果已超过一天，则无须操作；如果前一次投标时间未超过一天（判断方式：当前时间<投标时间+86 400 秒），则先通过调用前一次投标的 key 值取得其投标的小行星币数量（因为投标后该数量的小行星币已经由第三方托管，所以调用其值，用 keccak256 哈希函数计算后再赋值给 escrowKey），并且在第三方托管的小行星币余额上增加这一次投标的数量。

最后记录新投标并启动事件。调用 influenceStorage 的相关函数，将投标的 key 值、投标数量、投标地址存入存储空间，释放一个完成投标的事件。

```
/**
 * @dev Allow withdrawal of remaining escrow
 */
function withdrawEscrow() external {
  bytes32 key = keccak256(namespace, "escrow", msg.sender);
  uint escrowBalance = influenceStorage.getUInt(key);
  require(escrowBalance > 0);

  // Zero out balance in storage prior to transfering ETH
  influenceStorage.deleteUInt(key);
  msg.sender.transfer(escrowBalance);
}
```

【解读】本段程序代码定义了从第三方托管的账户撤回的功能。这是一个外

仅供非商业用途或交流学习使用

部函数，首先通过引用和 keccak256 转换取得第三方托管账户的 key 值，并转换为 uint 类型的余额，采用 require()函数检验来保证账户余额大于 0；然后调用 influneceStorage 的相关函数删除 key 的 uint 值，并将之前投标的 Token 转回。

```
  /**
   * @dev Lets the winner close the auction and retrieve the minted token
   * @param _asteroidId The Id of the asteroid bid on
   */
  function winAuction(uint _asteroidId) external returns (address) {
    uint bid;
    uint at;
    address by;
    (bid, at, by) = getCurrentBid(_asteroidId);

    // Make sure the sender is the high bidder and that we're at least
1 day after that bid
    require(msg.sender == by);
    require(now > (at + 86400));

    // Clear out storage
    influenceStorage.deleteUInt(keccak256(namespace, "bid", _asteroidId));
    influenceStorage.deleteUInt(keccak256(namespace, "bidTime",
_asteroidId));
    influenceStorage.deleteAddress(keccak256(namespace, "bidder",
_asteroidId));

    // Transfer winning bid to the house
    bytes32 houseKey = keccak256(namespace, "houseBalance");
    uint houseBalance = influenceStorage.getUInt(houseKey);
    influenceStorage.setUInt(houseKey, houseBalance + bid); // Do we
need SafeMath here?

    // Mint and send token to winner
    address tokenContractAddress = influenceStorage.getAddress
(keccak256("Contracts", "name", "AsteroidToken"));
    AsteroidTokenInterface tokenContract = AsteroidTokenInterface
(tokenContractAddress);
    tokenContract.mint(by, _asteroidId);
    emit AuctionWon(_asteroidId, by, bid, at);
  }
```

【解读】本段程序代码定义了赢得拍卖的函数 winAuction()。首先将当前投

仅供非商业用途或交流学习使用

标的信息赋值给 3 个变量：bid、at 和 by（分别代表投标数量、投标时间、投标账户），并利用 require()函数进行检验，确定当前投标值是最大值、投标时间已经超过一天，即一次投标的过程已经完成，胜者已经产生（投标值最大者为赢得投标者）。

然后调用 influenceStorage 的相关函数对内存中的对应数据进行清除。

接下来将赢得的投标值转账给赢家的存储账户，具体做法是在赢家原账户的余额上增加投标（bid）的数量。

随后要将这些 Token 完成铸币，发放给赢家。在取得投标合约的账户地址后，调用该合约的铸币函数 mint()，将投标数量的 Token 铸币后发放给赢家账户，最后释放出“赢得拍卖”事件。

```
  /**
   * @dev Withdraw from the house balance
   */
  function withdraw() external onlyOwner {
    bytes32 key = keccak256(namespace, "houseBalance");
    uint houseBalance = influenceStorage.getUInt(key);
    influenceStorage.deleteUInt(key);
    msg.sender.transfer(houseBalance);
  }
```

【解读】本段程序代码定义了从自有账户中撤回的函数 withdraw()，这是一个外部函数，从 onlyOwner()函数派生，其主要功能是在取得自有账户地址后，通过其余额进行转账操作。

```
  /**
   * @dev Returns the current bid, time, and bidder for a given asteroid
   * @param _asteroidId Token Id for the asteroid
   */
  function getCurrentBid(uint _asteroidId) public view returns (uint
bid, uint at, address by) {
    bid = influenceStorage.getUInt(keccak256(namespace, "bid",
_asteroidId));
    at = influenceStorage.getUInt(keccak256(namespace, "bidTime",
_asteroidId));
    by = influenceStorage.getAddress(keccak256(namespace, "bidder",
_asteroidId));
    return (bid, at, by);
  }
```

仅供非商业用途或交流学习使用

【解读】本段程序代码定义了公共函数 getCurrentBid()，用来返回当前投标的投标数量、投标时间和投标账户信息。

```
  /**
   * @dev Check escrow balance
   * @param _address Address to check for a balance
   */
  function getEscrow(address _address) public view returns (uint) {
    return  influenceStorage.getUInt(keccak256(namespace,  "escrow",
_address));
  }
```

【解读】本段程序代码用来进行第三方托管账户的检验，返回第三方托管账户的余额信息。

```
  /**
   * @dev Determins at which time an asteroid can be bid on based on its
ID (not seed)
   * @param _asteroidId Between 1 and TOTAL_ASTEROID. Same as param passed
to AsteroidFeatures.getAsteroidSeed
   */
  function availableForAuctionAt(uint _asteroidId) public view returns
(uint) {
    require(startTime != 0 && duration != 0); // Auction hasn't been set
yet
    uint asteroidsPerDay = TOTAL_ASTEROIDS / duration;
    uint dayAvailable = _asteroidId / asteroidsPerDay;

    /**
     * Round off the remainder into the last day to account for integer
division. Without this we end up
     * with the last several asteroids not available until the day after
the auction should end.
     */
    if (dayAvailable >= duration) dayAvailable = duration - 1;
    return dayAvailable * 1 days + startTime;
  }
}
```

【解读】本段程序代码的作用是判定什么时间的小行星可以用来拍卖。用 require()函数检验拍卖是否还没有完成（已经启动，持续时间不为 0），并确定每天的小行星数量（用总量除以天数）、当天可用的投标金额（用小行星币的余额除以每天的小行星数量）。

在最后一天要对剩余的小行星进行统计，如果不能被整除，就在当天结束拍卖。本函数返回从开始时间加上可用天数的数据，这就是可以进行拍卖的时间段。

9.4.4　游戏主功能：小行星代币

这一部分的源代码来自文件 AsteroidToken.sol，其主要功能是实现游戏中的小行星代币的形成过程。对源代码的解读如下。

1. 初始化和代币定义

```
pragma solidity ^0.4.21;

import "./InfluenceBase.sol";
import "./lib/SafeMath.sol";
import "./interfaces/InfluenceStorageInterface.sol";
```

【解读】Solidity 语言版本号声明，并导入程序所需的其他源程序，如库文件中的数学计算文件、interfaces 中的存储调用接口文件等。

```
/**
 * @dev Contract that manages each asteroid as an ERC721, non-fungible
token. Provided by OpenZeppelin.
 * Modified to use our eternal storage contract rather than internal state
to allow for upgrading.
 */
contract AsteroidToken is InfluenceBase {
  using SafeMath for uint;

  string public name = "Influence Asteroid Token";
  string public symbol = "INFAT";
  string public tokenURI;

  event Transfer(address indexed _from, address indexed _to, uint
_tokenId);
  event Approval(address indexed _owner, address indexed _approved, uint
_tokenId);
```

【解读】本段程序代码定义了合约 AsteroidToken，它是从 InfluenceBase 派生而来的，并使用 using for 语句将 SafeMath.sol 文件中的计算工具都附加到 uint

类型的计算中。随后定义了 3 个公共的字符串变量 name（名称）、symbol（简称）和 tokenURI（代币统一标识符），并将代币的名称命名为“Influence Asteroid Token”，简称代号为“INFAT”。

另外，还定义了两个事件：转账（Transfer）和确认（Approval）。

```
  /**
  * @dev Guarantees msg.sender is owner of the given token
  * @param _tokenId uint ID of the token to validate its ownership belongs
to msg.sender
  */
  modifier onlyOwnerOf(uint _tokenId) {
    require(ownerOf(_tokenId) == msg.sender);
    _;
  }
```

【解读】本段程序代码定义了修饰符 onlyOwnerOf()，用来确保 msg.sender 中的 Token 由其所有权人拥有，并保证_tokenId 与 msg.sender 保持一致，此时才能在“_;”处插入函数本体并执行该函数。

```
  /**
   * @dev Only owner and the initial auction contract should mint tokens
   */
  modifier onlyMinters {
    address auction = influenceStorage.getAddress(keccak256 ("Contracts",
"name", "AsteroidAuction"));
    require(msg.sender == owner || msg.sender == auction);
    _;
  }
```

【解读】本段程序代码定义了第二个修饰符 onlyMinters，用来确保让唯一的所有者、初始拍卖合约开始进行铸币，通过地址变量 auction、调用 influenceStorage 合约的函数取得拍卖合约的地址，然后采用 require()函数和“逻辑与”计算来确保两个条件同时具备，此时才能在“_;”处插入函数本体。

```
  function AsteroidToken(address _storage) InfluenceBase(_storage)
public {
    namespace = "AsteroidToken";
    version = 1;
  }
```

【解读】在 InfluenceBase()函数中派生出 AsteroidToken()函数来，命名其地

址空间为“AsteroidToken”，版本号为 1。

2. 代币发行

```
  /**
   * @dev Sets the token URI which points to a JSON adhering to ERC-721
metadata standard
   * @param _uri URL string
   */
  function setTokenURI(string _uri) external onlyOwner {
    tokenURI = _uri;
  }
```

【解读】本段程序代码定义了一个外部函数 setTokenURI()，其作用是设定代币（Token）的统一资源码（URI），以此来指向符合 ERC-721 元数据标准的 JSON 文件，同时定义了一个 URI 字符串。

```
  /**
  * @dev Gets the total amount of tokens stored by the contract
  * @return uint representing the total amount of tokens
  */
  function totalSupply() public view returns (uint) {
    return influenceStorage.getUInt(keccak256(namespace, "totalSupply"));
  }
```

【解读】本段程序代码定义了公共性质的函数 totalSupply()，其作用是取得合约中存储的总的 Token 数量，并返回代表 Token 总量的 uint 数值。

```
  /**
  * @dev Gets the balance of the specified address
  * @param _owner address to query the balance of
  * @return uint representing the amount owned by the passed address
  */
  function balanceOf(address _owner) public view returns (uint) {
    return influenceStorage.getUInt(keccak256(namespace,
"ownerBalance", _owner));
  }
```

【解读】这个函数用来获取特定地址的 Token 余额，其中_owner 的地址用来索引余额，返回值是一个 uint 整数，表示由该地址所拥有的 Token 余额。

```
  /**
  * @dev Gets the list of tokens owned by a given address
  * @param _owner address to query the tokens of
```

```
  * @return uint[] representing the list of tokens owned by the passed
address
  */
  function tokensOf(address _owner) public view returns (uint[]) {
    bytes32 key = keccak256(namespace, "ownedTokens", _owner);
    uint count = influenceStorage.countUIntSet(key);
    uint[] memory tokens = new uint[](count);

    for (uint i = 0; i < count; i++) {
      tokens[i] = influenceStorage.getUIntInSet(key, i);
    }

    return tokens;
  }
```

【解读】本段程序代码定义的函数是一个公共函数，同时也是一个视图函数，这表明函数的状态不会被修改。

tokensOf()函数的作用是获取由某个给定地址所拥有的 Token 的列表，其中 uint[]代表由合格的地址所拥有的 Token 列表。在该函数中定义了 key、count 及 memory 3 个参数，分别代表 Token 的键值、Token 的数量及 Token 列表。然后进行循环，调用 influenceStorage 的相关函数将 Token 列表中的数据赋给 token[]数组，并返回这个数组。

```
  /**
  * @dev Gets the owner of the specified token ID
  * @param _tokenId uint ID of the token to query the owner of
  * @return owner address currently marked as the owner of the given token ID
  */
  function ownerOf(uint _tokenId) public view returns (address) {
    address owner = influenceStorage.getAddress(keccak256(namespace,
"tokenOwner", _tokenId));
    require(owner != address(0));
    return owner;
  }
```

【解读】本段程序代码定义了公共的视图函数 ownerOf()，用来获取特定 token ID 的所有者信息，返回当前标注为“所有者”的地址。执行过程中需要先确认所有者与起始地址是否不同，只有在二者不同的情况下才会返回 owner。

```
  /**
   * @dev Not a part of ERC721 standard. Returns whether this token has
been minted yet (eventually all
   * within the range of 1 to TOTAL_ASTEROIDS will)
   * @param _tokenId Asteroid ID up to TOTAL_ASTEROIDS, as as in
AsteroidFeatures.getAsteroidSeed
   */
  function isTokenMinted(uint _tokenId) public view returns (bool) {
    address owner = influenceStorage.getAddress(keccak256(namespace,
"tokenOwner", _tokenId));
    return owner != address(0);
  }
```

【解读】本段程序代码定义了判断 Token 是否已用的函数 isTokenMinted()，属性代码有公共的、视图式的。所有者的地址从 influenceStorage 处调用并索引，然后返回 owner。

```
  /**
   * @dev Gets the approved address to take ownership of a given token ID
   * @param _tokenId uint ID of the token to query the approval of
   * @return address currently approved to take ownership of the given
token ID
   */
  function approvedFor(uint _tokenId) public view returns (address) {
    address approved = influenceStorage.getAddress(keccak256
(namespace, "tokenApprovals", _tokenId));
    return approved;
  }
```

【解读】本段程序代码定义的函数同样是一个公共类型的视图函数，用来取得某些已被认可的 Token 的地址，从而取得给定 token ID 的所有权。这个函数的参数是一个地址，通过该地址调用给定的 Token，赋值给系统认可的地址，并返回这个系统认可的地址。

3. 代币转账交易

```
  /**
  * @dev Transfers the ownership of a given token ID to another address
  * @param _to address to receive the ownership of the given token ID
  * @param _tokenId uint ID of the token to be transferred
  */
  function transfer(address _to, uint _tokenId) public onlyOwnerOf
(_tokenId) {
```

```
    clearApprovalAndTransfer(msg.sender, _to, _tokenId);
  }
```

【解读】对给定的 Token 与另一个地址之间进行所有权转让，地址变量_to 用来接收给定 Token 的所有权转让，_tokenId 是转出的 Token 数量。这个函数也是 onlyOwnerof()函数的一个派生函数。

```
  /**
  * @dev Approves another address to claim for the ownership of the given
token ID
  * @param _to address to be approved for the given token ID
  * @param _tokenId uint ID of the token to be approved
  */
  function approve(address _to, uint _tokenId) public onlyOwnerOf
(_tokenId) {
    address owner = ownerOf(_tokenId);
    require(_to != owner);

    if (approvedFor(_tokenId) != 0 || _to != 0) {
      influenceStorage.setAddress(keccak256(namespace,
"tokenApprovals", _tokenId), _to);
      emit Approval(owner, _to, _tokenId);
    }
  }
```

【解读】本段程序代码定义了一个公共函数 approve()，从 onlyOwnerOf()派生而来，作用是对于给定的 Token，许可另一个地址对其主张所有权。地址变量_to 用来授予给定的 Token，_tokenId 是授予的 Token 数量，同时用 require()函数检验前提条件：所有者账户与授予账户不能相同。当许可行为的条件得到满足后（授予 Token 的数量不为 0，地址不为 0），调用 influenceStorage 合约的函数进行许可，并释放出一个 Approval 事件。

```
  /**
  * @dev Claims the ownership of a given token ID
  * @param _tokenId uint ID of the token being claimed by the msg.sender
  */
  function takeOwnership(uint _tokenId) public {
    require(isApprovedFor(msg.sender, _tokenId));
    clearApprovalAndTransfer(ownerOf(_tokenId), msg.sender, _tokenId);
  }
```

【解读】本段程序代码定义了一个公共函数 takeOwnership()，其作用是对给定数量的 Token 主张所有权，先进行 require()条件检验，即将地址、Token 数量进行许可，然后对账户进行转账并完成授予。

```
/**
* @dev Mint token function
* @param _to The address that will own the minted token
* @param _tokenId uint ID of the token to be minted by the msg.sender
*/
function mint(address _to, uint _tokenId) public onlyMinters {
  require(_to != address(0));
  addToken(_to, _tokenId);
  emit Transfer(0x0, _to, _tokenId);
}
```

【解读】这是一个公共的铸币函数，它从 onlyMinters 派生而来，_to 地址用来接收铸出的币，_tokenId 则是由 msg.sender 铸币的数量。在函数的执行过程中，先用 require()检验铸币接收地址与初始地址是否不同，然后在_to 的地址上增加 Token 数量，最后释放出一个转账事件，完成铸币。

```
/**
 * @dev Tells whether the address is approved for the given token ID
or not
 * This function is not private so it can be extended in further
implementations like the operatable ERC721
 * @param _owner address of the owner to query the approval of
 * @param _tokenId uint ID of the token to query the approval of
 * @return bool whether the msg.sender is approved for the given token
ID or not
 */
function isApprovedFor(address _owner, uint _tokenId) internal view
returns (bool) {
  return approvedFor(_tokenId) == _owner;
}
```

【解读】本段程序代码定义了一个内部函数 isApprovedFor()，用来判断某个地址是否被授予给定的 Token，返回值是一个布尔值（bool），1 表示授予成功，0 表示未能被授予。

```
/**
* @dev Internal function to clear current approval and transfer the
ownership of a given token ID
```

```
  * @param _from address which you want to send tokens from
  * @param _to address which you want to transfer the token to
  * @param _tokenId uint ID of the token to be transferred
  */
  function clearApprovalAndTransfer(address _from, address _to, uint
_tokenId) internal {
    require(_to != address(0));
    require(_to != ownerOf(_tokenId));
    require(ownerOf(_tokenId) == _from);

    // Clear out approval (which may be empty) and swap token, fire event
    clearApproval(_from, _tokenId);
    removeToken(_from, _tokenId);
    addToken(_to, _tokenId);
    emit Transfer(_from, _to, _tokenId);
  }
```

【解读】本段程序代码定义了一个内部函数 clearApprovalAndTransfer()，这个函数在本合约中已经多次被其他函数引用，它的功能是清除当前的 Token 授权，并将给定 Token 的所有权进行转移，转出地址为_from，转到的地址为_to，转让的数量为_tokenId。在进行清除和转让之前先进行 require()检验：_to 转到地址与初始地址不同，转到账户与原始所有者账户不同，转出账户的所有者就是 Token 的拥有者。然后清除授权，移动 Token，增加转到账户的 Token 数量，最后释放出一个转账事件。

```
  /**
  * @dev Internal function to clear current approval of a given token ID
  * @param _tokenId uint ID of the token to be transferred
  */
  function clearApproval(address _owner, uint _tokenId) private {
    require(ownerOf(_tokenId) == _owner);
    influenceStorage.deleteAddress(keccak256(namespace,
"tokenApprovals", _tokenId));
    emit Approval(_owner, 0, _tokenId);
  }
```

【解读】这是一个私有的清除许可函数，仅在本合约中可见，作用与上一个清除许可函数相似，但它是通过调用 influenceStorage 合约中的 deleteAddress()函数来实现的。



4. 代币数量更新

```
  /**
   * @dev Internal function to add a token ID to the list of a given address
   * @param _to address representing the new owner of the given token ID
   * @param _tokenId uint ID of the token to be added to the tokens list
of the given address
   */
  function addToken(address _to, uint _tokenId) private {
    require(_tokenId > 0 && _tokenId <= TOTAL_ASTEROIDS);
    bytes32 ownerKey = keccak256(namespace, "tokenOwner", _tokenId);
    address tokenOwner = influenceStorage.getAddress(ownerKey);
    require(tokenOwner == address(0));

    // Assign new owner and add token to owned set
    influenceStorage.setAddress(ownerKey, _to);
    influenceStorage.addUIntToSet(keccak256(namespace, "ownedTokens",
_to), _tokenId);

    // Update ownership count
    bytes32 balanceKey = keccak256(namespace, "ownerBalance", _to);
    uint ownerBalance = influenceStorage.getUInt(balanceKey);
    influenceStorage.setUInt(balanceKey, ownerBalance.add(1));

    // Update total supply
    bytes32 supplyKey = keccak256(namespace, "totalSupply");
    uint totalTokens = influenceStorage.getUInt(supplyKey);
    influenceStorage.setUInt(supplyKey, totalTokens.add(1));
  }
```

【解读】这也是一个私有的函数 addToken()，在本合约中可见，在其他合约、派生合约中均不可见，其作用是把 Token 增加到给定的地址列表中。这个函数有两个参数：_to 是给定的地址列表；_tokenId 是要增加的 Token 数量。

执行过程如下：

（1）进行 require()检验，需满足“0<增加的 Token 数量<Token 总量”的条件，同时将这一批 Token 的地址从存储器中取出，调用 influenceStorage 合约中的函数将其转换为持有者地址，这个地址必须与初始账户地址相同。

（2）将要转到的账户地址赋给初始账户，并将账户内的 Token 数量加上 _tokenId 的数量。



（3）更新“所有权的计数”，即对于所有权变动次数的计数，将“转到账户”与“所有者余额”混合哈希，并转换为 uint 数据，增加 1 后，将这个值再赋给账户索引键值，使所有权变动次数得以实时更新。

（4）更新 Token 总供给量的变动次数。定义总供给量存储空间的哈希值为总供给量索引键值，转换为 uint 类型后增加 1，返回这一转换后的 uint 类型的索引键值，完成对总供给变动次数的实时更新。

```
  /**
  * @dev Internal function to remove a token ID from the list of a given
address
  * @param _from address representing the previous owner of the given
token ID
  * @param _tokenId uint ID of the token to be removed from the tokens
list of the given address
  */
  function removeToken(address _from, uint _tokenId) private {
    require(ownerOf(_tokenId) == _from);
    influenceStorage.deleteAddress(keccak256(namespace, "tokenOwner",
_tokenId));
    influenceStorage.deleteUIntInSet(keccak256(namespace,
"ownedTokens", _from), _tokenId);

    // Update ownership count
    bytes32 balanceKey = keccak256(namespace, "ownerBalance", _from);
    uint ownerBalance = influenceStorage.getUInt(balanceKey);
    influenceStorage.setUInt(balanceKey, ownerBalance.sub(1));

    // Update total supply
    bytes32 supplyKey = keccak256(namespace, "totalSupply");
    uint totalTokens = influenceStorage.getUInt(supplyKey);
    influenceStorage.setUInt(supplyKey, totalTokens.sub(1));
  }
}
```

【解读】本段程序代码定义了一个私有函数 removeToken()，参数有两个：_from 是前一个所有者的地址；_tokenId 是要从给定的地址列表中移除的 Token 数量。这个函数的作用是从给定的地址列表中移除 Token。

执行过程如下：

（1）进行 require()检验，确保移除的地址对数量为_tokenId 的 Token 是拥



有所有权的。

（2）调用 influenceStorage 合约中的相关函数，清除对应 Token 的数量和地址空间。

（3）分别对“所有权变动次数”“总供给量变动次数”进行更新，执行过程与 addToken()函数的执行过程基本相同，唯一的不同是：由于函数实现的作用是移除 Token，所以在此处更新时调用的计算函数为减法。

以上对源程序 AsteroidToken.sol 的全部内容进行了解读，从中可以看出，开发者灵活运用了 Solidity 语言的基本元素，实现了游戏所发行的代币——小行星币的铸币、转账和数量的更新等各种功能，这也是以太坊及区块链游戏的特点之一，即通过合约来实现游戏币的自动分发。

9.4.5　小结

从 GitHub 中可以看出，每个以太坊应用程序都有大量的 Solidity 源代码，用来实现不同的功能。限于篇幅，本书不可能对其他源程序文件进行一一解读，只能通过对一些典型的 Solidity 源代码进行解读，帮助读者触类旁通、举一反三，获得对于 Solidity 语言的清晰认知。

在解读源代码的过程中，读者也可以发现，Solidity 语言用来开发以太坊应用程序是一种比较友好的语言，它语义明确、结构清晰，变量、参数、函数、合约的逻辑层次明显，实现功能的方式灵活多样，还能对存储空间进行操作，能够很好地满足当前对以太坊各类 DApp 编程的需要。

另外，Solidity 语言也在根据全球开发人员应用的实践经验不断进行迭代，对现有的功能进行大幅优化和提升。因此，深入掌握 Solidity 语言，对于以太坊 DApp 开发者而言是一件十分有意义的事情。



后记

深入区块链，用技术改变未来

从2017年以来，区块链成为当今时代极其火热的主题，无数的人才向这个领域进发，无数的资金向这个领域集中，全社会的注意力都被吸引过来，甚至在社交网络上关于区块链的交流彻夜不眠。随着以比特币、以太币等为代表的虚拟货币的价值剧烈波动，区块链也引发了全球各地政府对其监管的提升。内部和外部环境的变化令区块链这个领域也不断发生分裂：有的人只为了投机炒作虚拟货币；有的人则坚信区块链将成为颠覆性的力量，是为每个网络节点赋能的技术。

从本质上来看，区块链是一项推动人类经济关系变革的技术，它本身并非革命性的创新技术，而是在前人的各类技术开发（如密码学、分布式网络、编程语言等）的基础上进行的一种集成创新，从而形成了一种全新的经济关系：人与人之间进行交易不再需要通过中心化的机构来提供信任，而且产生的交易记录将以最广泛的形式进行分布，且无法更改。这种新的经济关系将对人类社会现有的经济生态、社会生态等产生深远的影响。虽然这一技术还处在早期的探索阶段，还存在种种不完善的地方，但是不会改变它本身所具有的这种新型经济关系而自带的颠覆性影响。

目前，区块链行业的从业人员普遍认为，如果用互联网的发展历程类比，那么现在区块链所处的阶段相当于 2000 年前后全球互联网行业蓬勃发展的阶段，由于发展迅猛，不可避免地带来了泡沫和浮躁，但也说明这个行业正在步入快车道，在广度和深度两个维度上不断向社会的纵深拓展。在这样一个时期，



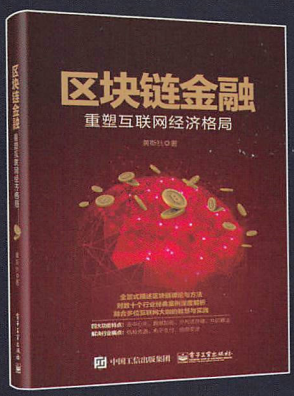
与其旁观、坐而论道，不如打开心胸，拥抱趋势和潮流，投入区块链这个领域中，追随技术发展的洪流。反之，空洞的评判、无谓的争论其实并无意义，因为在当前阶段，区块链这个领域还存在大量的无人区，仅仅套用现有的、传统的规则和价值观，实际上是不可行的。一句话：面对区块链，最重要的是深入了解、积极实践，在不断的迭代过程中掌握区块链技术。

认知区块链，需要足够的理性精神。它毕竟是网络科技的一个分支，它的基础还是各种网络技术。虚拟货币并不是区块链的全部，而只是一种辅助性的手段，用来实现信任的转移，以及维护区块链网络的运转。所以，认识和了解区块链，依然要回归技术层面，了解其原理和应用，才能“知其所以然”。

为此，我们编写了这样一本书。它从区块链的发展过程和表象入手，层层深入，一直深入具体的技术层面，揭开区块链的面纱。在此基础上，结合全社会对于从“区块链 1.0”的比特币发展到“区块链 2.0”的以太坊的趋势，以及以太坊作为开放的公有链、以智能合约为特点的 DApp 在全球范围内快速发展的认识，重点引入了以太坊的知识，以及对采用 Solidity 语言开发以太坊应用程序的介绍。当然，作者水平有限，本书必然存在不足和疏漏之外，欢迎读者不吝赐教。

总的来讲，无论虚拟货币如何变换和波动，都只是区块链技术发展浪潮中的泡沫，真正的激流依然是核心技术。只有理解了区块链的基础概念，并通过以太坊编程进入开发领域，我们才有可能通过技术改变未来。





ISBN：978-7-121-34876-1
定价：65.00元

为读者出好书，传播文字的思想价值；

为作者做好书，提升智慧的商品价值。

美迪出版

团购电话：010-88254062

预知新书信息、交流投稿、邮购团购

请发邮件至：xyan@phei.com.cn

本书在全国各大新华书店、书城、均有销售

浏览请登录：www.phei.com.cn

新浪微博@美迪出版

豆瓣：美迪出版

区块链2.0实战

第1章 区块链基础

定义区块链
区块链的发展经过与现状
区块链在中国的发展现状
区块链在中国的发展前景

第2章 区块链的商业价值

区块链在银行业的应用
区块链与电子商务
区块链在法律行业的应用
区块链在影音娱乐和媒体行业的应用
区块链在医疗行业的应用

第3章 区块链技术原理

比特币白皮书
比特币区块
比特币挖矿
区块链系统框架
分布式网络
广播与验证机制

第4章 区块链共识机制

工作量证明机制
权益证明机制
授权股权证明机制

第5章 区块链数据结构

区块链的数据组成
区块链哈希算法
椭圆曲线加密算法
区块链交易机制
区块链造链机制
区块数据解读

第6章 以太坊

以太坊的创立和发展
以太坊账户
智能合约
以太坊挖矿
以太坊区块

第7章 以太坊应用开发基础

以太坊开发环境的建立
Geth开发环境
轻节点模式
Remix模式
Mist浏览器
MetaMask应用
Solidity语言

第8章 Solidity开发基础

Solidity语言的数据类型
Solidity语言的表达式和控制结构
以太坊合约
Solidity汇编

第9章 采用Solidity语言开发以太坊游戏

以太坊游戏的特点
以太坊游戏开发准备
以太坊游戏案例-代码框架
以太坊游戏案例-源代码解读

上架建议：计算机、网络与互联网

ISBN 978-7-121-34877-8

9 787121 348778 >

定价：78.00元

PHEI 美迪出版

责任编辑：张　毅
咨询服务：010-57565890
投稿邮箱：meidipub@phei.com.cn